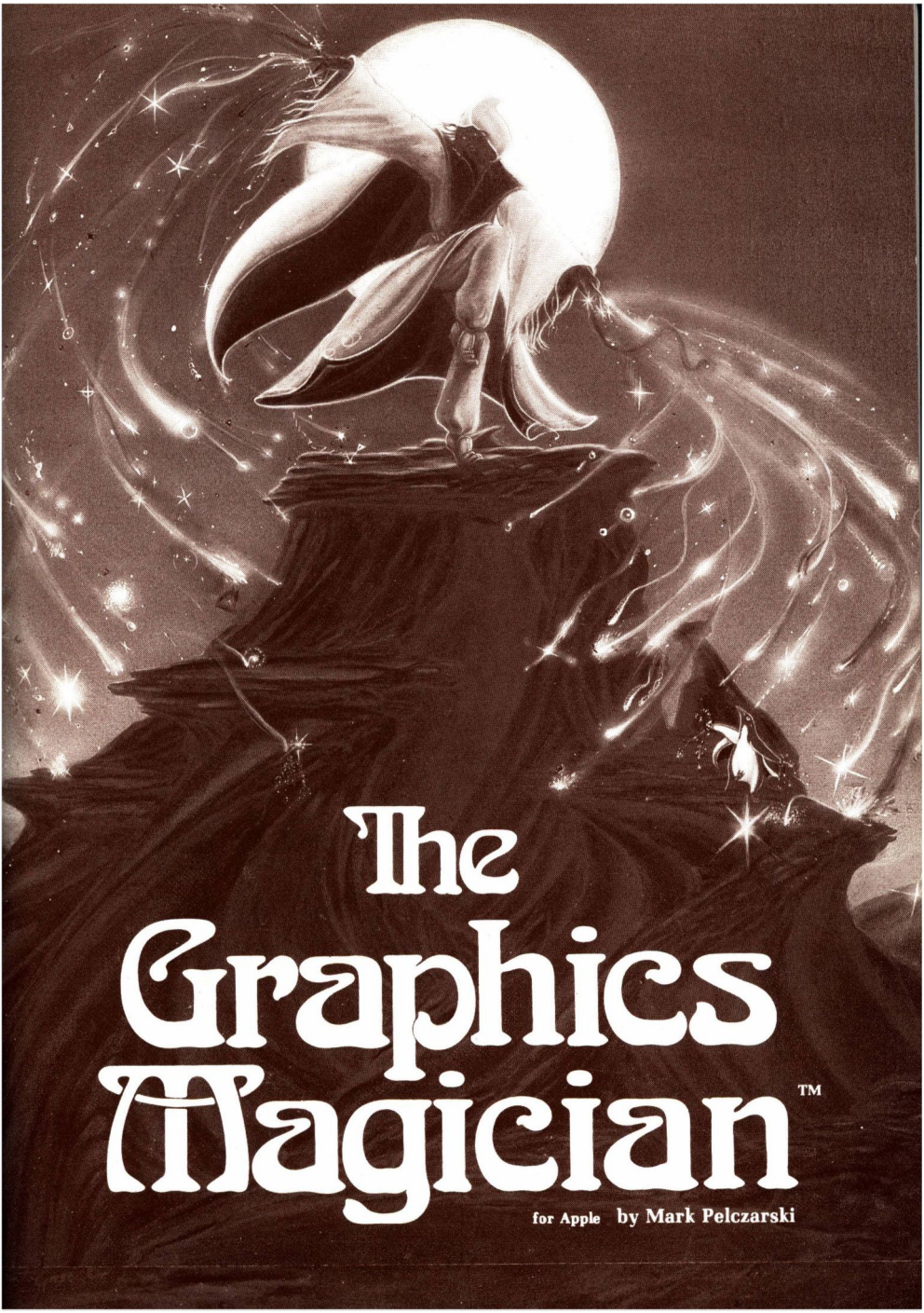

The
Graphics
Magician™
for Apple by Mark Pelczarski

The Graphics Magician™

penguin software™

the graphics people

P.O. Box 311, Geneva, IL 60134 (312) 232-1984

The enclosed product is supplied on a disk that is NOT copy-protected. It is our intent to make this product as useful as possible, and we feel that with applications software the ability to make your own backup copies is a great asset. We ask that you not abuse our intentions by making copies for others, and by not copying this manual. The result of such activity only helps promote the unfortunate existing situation of most software being protected, and, in our opinions, less usable. We hope that the commercial success of non-protected products such as this one will signal other publishers that copy-protection is not necessary for a product's survival and help reverse the trend in protected applications software. At Penguin, through policies such as this and our pricing, we are trying to look out for your best interests. Please help us.

We hope you find many hours of enjoyment and use in this package.

51832

Sincerely,

Mark Pelczarski

Mark Pelczarski
President, Penguin Software

The Apple version of **The Graphics Magician** is written by the following people, all of whom either contributed writing, routines, or ideas to the current version: Mark Pelczarski, David Lubar, Chris Jochumson, Larry Lathrop, Rob Engels, Dallas Snell, Ame Flynn, and Jon Niedfeldt.

Table of Contents

Part Two—The Picture System

Chapter 1 - Introduction

The Graphics Magician is a set of graphics editors and machine language routines that help you easily put professional, state-of-the-art graphics and animation in your own programs. Included are routines that help you design and control fast animation of many independent objects (as in most arcade games), draw and recreate very compact, multicolored pictures (great for use in adventure games and educational software), and easily put text on the graphics screen. To fully use these routines, you will have to do some programming.* This may be as little as a 3-line program in Basic, and examples are given throughout this manual. **The Graphics Magician** is meant to be a programmers' tool. It's designed to be easy to use for the beginning Basic programmer, yet it has a vast amount of flexibility built in for even the most advanced machine language programmers. **The Graphics Magician** routines included in this package are being used in dozens of commercially marketed software packages, produced by many of the most well-known publishers in the industry. And with **The Graphics Magician** available on several leading microcomputers, designers will find that most of the graphics work done on one machine can be easily transferred to several others, saving long hours of duplicated work, and in some cases making software portable in ways never before possible.

When learning to use **The Graphics Magician,** start simply. First of all, sit down with your manual AND your computer. This is computer software, not a book. You'll learn it best by using it while reading about it. Since **Graphics Magician** has a lot of features included, it is tempting to want to understand them all immediately. For animation, we recommend that you first create a simple, black and white shape with the shape editor, a simple path with the path editor, then put them together in the animation editor and test the animation to see what happens. Then, if you want, save the animation file and try a simple 3 or 4 line Basic program to see how your programs will control the animation. Chapter 2 has a tutorial for doing precisely those steps. For the picture system, just try some scribbling with lines and fills. Test the "redraw" command to see how your picture is recreated. Then save it, and again write a 3 or 4 line Basic program to see how you load and draw the picture yourself. Only then should you go on and start playing with the other features and functions available. As with any programming, experiment and practice...

*If your needs are more for designing static computer graphics or art, 3-D design, slide creation, title pages, and other such applications, we recommend our **Complete Graphics System,** which does not require programming background, and provides a lot more flexibility for the above needs.

A Little Background

A little background on the development of **Graphics Magician** may explain the inclusion of some things in this manual, and perhaps relieve some anxieties about not always understanding why they are there and what they are for. When the original version was written, we knew what capabilities we wanted and what the results should look like. With the animation system, in particular, we knew what should happen, but actually had to build all the modules before the real animation could be tested. When it all came together, we had the capabilities we wanted, but there were also a lot of things we had built into the code that seemed like they'd be interesting to some programmer, someday. Pointers were included to things that would let you change paths, switch shapes, pick up an object and set it down across the screen, control object movement with human actions, and even access the most basic of the plotting routines we used. So we documented all those things that could be controlled, even if we weren't sure exactly why one would want to, and put them in the manual. It's sort of like building a hammer. You don't know all of the uses to which it will be put, or the things that could be made with it, but you can tell people as much as you know about it.

Well, a lot of the things we documented have been put to neat uses (and we've got many of those examples and ideas included in the manual now), and others haven't really been needed. There have been a lot of things done with **Graphics Magician** that we didn't imagine in our original design, and uses to which it's been put that we never anticipated. So don't feel you have to understand everything in this manual and its immediate use. Someday you may discover your own.

Input Devices

The Graphics Magician uses keyboard for accurate cursor control in animation design. In the picture editor, you may use joystick, paddles, trackball, mouse, Koala Pad™, or any other joystick-compatible device that returns x,y values directly through the paddle I/O port.

The Apple Graphics Tablet works with the tablet version. **The Graphics Magician** comes configured for use with the tablet in slot 5. To change this, run APPLE TABLET INIT, which is on the **Graphics Magician** disk. With the Apple tablet, pressing the pen down corresponds to using joystick button 0 and the Return Key on the keyboard functions as joystick button 1.

The Graphics Magician also works with the Houston Instruments HiPad™ model DT11A or the DT11 with a DT11-128 parallel interface card. Before using **The Graphics Magician,** run HIPAD CALIBRATE, which is on your **Graphics Magician** disk. To calibrate, the HiPad must be in stream mode.

Backup Copies

The Graphics Magician is provided on a copyable disk, and all the routines are accessible by your own programs. We strongly recommend that the first thing you do is make a backup copy or two and store the original in a safe place.

A registration number is written on your **Graphics Magician** disk and stamped on the inside cover of your manual. If you call with questions regarding use of this product, please be prepared to provide this number. Also remember to send in your registration card so that we may notify you of any new versions or updates.

Licensing

Any of the routines enclosed may be freely used in your own programs. If the routines or facsimiles appear in another product for sale, there is no fee, but we do require that a license be obtained from Penguin Software stating that you have permission to use the copyrighted routines. Note that **The Graphics Magician** is or will be available for several different microcomputers. In writing the different versions, the routines were made as compatible as possible, so much of the work you do on one computer could be easily transferred to another, if you desire. Some of the basic transfer routines are included in this package. Also, please consider Penguin Software as a possible publisher for your works.

Getting Started

First, using COPYA from your Apple System Master, or any other copy program, make one or two copies of each side of your **Graphics Magician** master disk (the back side contains a set of sample shape and animation files). Put the master away in a safe place, and use the copies in everyday work.

To use the Picture System, you MUST be working with a BACKUP copy made on a NOTCHED disk. This is because the program occasionally must write to the disk to temporarily save information. Attempting to use your write-protected master disk will result in a "Write Protect Error" and the program will break.

Next, initialize a blank disk or two for use as data disks. **The Graphics Magician** disk is close to full. The shapes, paths, animation files, and pictures you create should be saved on a separate data disk. To initialize a disk, make sure you start with a disk that contains nothing important already on it (it will be erased). Then "boot" your Apple System Master (put it in the drive and turn on the computer). After the disk stops, type "NEW" and press Return. Then insert your blank disk and type "INIT HELLO" and press Return. The disk will whir for a while, and when it stops you have an initialized disk.

Now boot your **Graphics Magician** copy. You will be presented with a list of choices in parentheses. This is the "menu" screen. When done using any of the modules listed in the choices, you will always be returned to this page. Note that throughout this manual, for ease of reading and understanding, single key choices will be listed with their meanings. Most of the choices in **Graphics Magician** require only a single keypress, but expressing them in the form "(**S**)hape editor" instead of "S" helps you follow the meanings of each a little more easily.

The Graphics Magician comes ready to use on an Apple with one disk drive. If you have one drive, you will have to switch disks when you read and save your shape, path, animation, and picture files to your data disk, and again each time you are done with a module and wish to return to the menu screen. (The copy of the back side of **The Graphics Magician** is also treated as a data disk). If you have more than one drive, you can set **The Graphics Magician** so that it knows where the master and data disks are located. Normally, you should put the master in drive 1 and the data disk in drive 2. To set this, choose "(**M**)odify disk access" from the menu. You will be asked for the locations of the master disk and data disk. Type "D1" and "D2". Make sure the "D" is included, or your computer will get confused. You may also configure for multiple slots or multiple volumes (on a hard disk) with specifications such as "S5" or "V21". Check with your dealer if you have questions about use of other such devices.

Now you're ready to go. Decide if you'd rather start with the animation editors or the picture editor, and skip to the appropriate section of this manual. Remember, don't try too much too fast. Try simple pictures and shapes, then experiment with the options in each editor, one at a time.

Part One - The Animation System

Chapter 2 - A Quick Tutorial

The animation system is made up of three editors: the shape editor, the path editor, and the animation editor, and a documentation utility that lets you print all the important addresses and information about finished animation files. The general procedure for creating animation is to create the shapes you need in the shape editor, draw paths that they will move in with the path editor, then assign shapes to paths and starting locations in the animation editor. The animation editor will save a complete animation file consisting of the machine language animation routines, shapes, paths, and associated tables. From your own program, you then load the animation file and call it each time you want to advance one step in the animation. Your program can check the tables after each animation call for any interaction between objects, or can change paths or shapes, and so on.

Creating an Animation File

Let's start from the beginning with a simple shape and path, and see how it's done. First, from the menu, choose "(**S**)hape editor".

A Simple Shape

When the shape editor is loaded, it first asks a few quick questions. For now, press Return for each to get the standard settings. When it asks for WIDTH and HEIGHT, type 10, then Return, for each.

The screen will then show 7 identical sections, each bordered by 4 dots and topped with an orange bar. A blinking cursor appears in each. At the bottom is a list of single letter commands that can be used, and other status indicators, including the x,y cursor positioning. The back pages of this manual contain a reference sheet for all the commands, but the ones you should first be concerned with are **I, J, K,** and **M.** Press each a few times and watch the flashing cursors and the x,y positioning. I is up, J is left, K is right, and M is down. Next, press "**Q**". "Q" locks the plotting of the cursors on. Move the cursor around now and notice that it leaves an identical path in each of the sections.

Scribble for a little while, then press "(**A**)nimate". This does a quick animation of your shape across the bottom of the screen. If you have a joystick or paddles connected, play with the settings... they control the speed. Pressing any key gets you back to edit mode.

We'll keep this shape for now. You'll learn the other options later. Press "(**O**)ptions", which gives you the disk options. If you are using only one drive, put in your data disk. Press "(**S**)ave", and when you are asked for a name, type in the name BLOB, which is probably close to what the first try looks like, then press Return. After the disk stops, put your master back in and press "(**Q**)uit". It will ask if you want to lose the current shape. Since you've already saved it on disk, press "(**Y**)es." The menu screen will appear in a few seconds.

Now, A Path

From the menu, choose "(**P**)ath editor." You'll be asked if you want to load a background screen. Since you don't even have one, type "(**N**)o."

Now you see a dot in the middle of the screen, and a list of choices at the bottom, similar to those in the shape editor. A message says to move the cursor to the start. Its present location is as good as anywhere for now, so just type "(**S**)tart." Now play with the keys **U, I, O, J, K, N, M,** and ",". They work similarly to the IJKM keys in the shape editor, except now you have the diagonals, too. (Note that the keys move in the direction of their arrangement on the keyboard). You are drawing a path for your shape to follow. Draw for a while, and then try to bring the last point somewhere near the first. Then type "(**S**)ave" and name your path SCRIBBLE. If you are using one drive, make sure your data disk is in. When it's done, replace your master disk, then type "(**Q**)uit" and "**Y**" (yes, the path can now be lost since it's on disk). Back to the menu!

On to the Animation

Now press "(**A**)nimation editor". In the editor, you are given a text screen with the options listed. Insert your data disk, and press "(**L**)oad". The program will ask what you want to load. Press "(**S**)hape", then type in the name of your shape, BLOB. After it loads, press "(**L**)oad" again, and "(**P**)ath" to load your path, SCRIBBLE.

Now that everything's loaded, you can create an object. (For definition's sake, an "object" is the thing that animates. It has a shape, a path to follow, and a starting location. Several objects can have the same shape or path, and likewise, they can change shapes or paths while animating.) Press "(**O**)bject create".

A quick note here about numbering: everything starts with number 0, so the BLOB is shape 0, and SCRIBBLE is path 0, and likewise, the first object you create is object 0. It would have been just as easy to start with 1, but later the 0 turns out to be more convenient for programming.

Now just fill in the blanks. Use shape #0. (If you just press Return, you'll find that shape 0 is "BLO", the first 3 letters of the actual name). For starting coordinates, give 100 for both x and y. Then, for path A, type 0 (SCRIBBLE). For path B, you can press "**R**" to repeat the previous path (setting up a loop). Then you are asked which step of the path to start in. Type 0, although you can actually start your object anywhere in the path.

You're done. Press "(**H**)i-res clear" to clear the graphics screen, then press "(**A**)nimate". The paddle or joystick will once again control the speed of your object, moving in the path you drew. Any key will return to the options.

For kicks, create another object, using "(**O**)bject create" again. Give it shape 0 and path 0, with "(**R**)epeat", again. Except start this one at coordinate 50,70. When you get back to the options, try "(**A**)nimate" to see what happens.

Finally, "(**S**)ave" your animation file with the name FIRST. The last thing we'll try is to load and run the animation from a Basic program, then we'll take a detailed look at the options available in each of the editors, and how and why the animation works. But first, type "(**Q**)uit" from the animation choices, put your master disk back in, and verify with a "(**Y**)es" to get back to the menu.

Putting it in a Program

From the menu, type "(**Q**)uit", and it will put you back in Applesoft, with the square bracket (]) prompt at the bottom of the screen. You're now ready to type in a Basic program that will use the animation file you just created. Type in the 4-line program in listing 2.1.

```
10 HGR : POKE - 16302,0
20 PRINT CHR$ (4);"BLOAD FIRST.ANM"
30 CALL 36864
40 GOTO 30
```

Listing 2.1 - Animation from Basic

The first statement sets the high resolution graphics display. The POKE also clears the bottom text lines that normally appear (try it without using the POKE to see what happens). The second statement does a binary load (BLOAD) of your animation file, FIRST.ANM. Note that the suffix ".ANM" was added by **Graphics Magician** when it was saved, telling you that this is an animation file. The third line, 30, is a CALL to the animation routine. Each time this CALL is executed, the animation advances one step. Line 40 loops back to line 30, so that the animation just cycles through its steps indefinitely.

RUN the program. You'll see your animation again, just as it looked when you tested it in the animation editor, except now it's from your own program! The exciting part is that by adding extra steps between what are now lines 30 and 40, you can exercise any kind of control over the animation that you want... joystick control of objects, random changes in paths, activating and deactivating objects, and on and on...!

Examples of those controls and lists of all the information available are given in chapters 9 and 10. First there are chapters on the animation types available and why they work, and on each of the three editors you've just used. You don't have to master any one of chapters 3-10 before moving to another. We recommend that you skim them and pick up and try parts of interest before trying to understand all the options available. There's plenty of room to experiment and lots of time in which to learn.

Chapter 3 - The Animation Routines

Technical Background of Hi-res Graphics

The design of the animation system is based on a special type of shape called a "pre-shifted" shape, which is actually a descendent of character graphics. The graphics screen is a display of an area of approximately 8,000 bytes of the computer's memory. Each byte is 8 bits, or ON or OFF's, of which 7 of those dots are displayed. The bytes are arranged so that there are 40 across the screen, arranged horizontally, so there are 280 (40 times 7) dots across the screen left to right. Vertically, there are 192 bytes "stacked" on top of one another, so the screen is 192 dots tall. See figure 3.1 for a cutaway portion of what this looks like.

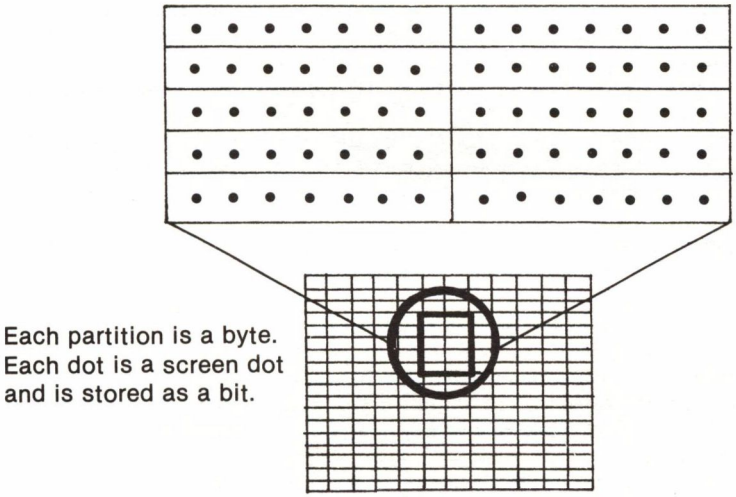


Figure 3.1 - A Portion of the Graphics Screen

Character graphics involve storing a certain bit pattern in a set of bytes, so that when those bytes are stored in screen memory, a shape is displayed. That is how text appears on the screen. Each normal sized character is 7 dots wide (1 byte) and 8 dots tall. So a set of 8 bytes defines each character. When those 8 bytes are stored in the proper sequence on the screen, a letter, such as the letter "A", appears.

The problem with using character graphics for animation is that the fastest way to move things around is by bytes, not bits. Vertically, it's easy to move a shape up or down one dot because you can just move it down one byte. Horizontally, though, if you move it one byte over, the shape moves 7 dots! That doesn't allow much control over smooth animation. To move in smaller increments, the bits that define the shape actually have to be shifted over within the bytes, which is both time-consuming and messy.

The solution is "pre-shifted" shapes (which have no relation at all to regular Applesoft shapes). For every shape created, seven facsimiles are actually stored as bit patterns in memory. The bit patterns are the same, except each is moved over one dot with respect to the byte boundaries. That way fast character graphics can be performed with totally flexible

horizontal movement. The animation routine simply checks the x-coordinate, then selects the bit pattern that fits the proper byte boundary. This is the basis for the fast animation you see in virtually all of the good, professional arcade games.

Animation Types

The animation routines in **Graphics Magician** are all based around this concept of pre-shifted shapes. The most basic PLOT routine does the necessary lookup that selects the proper bit pattern to put on the screen. One level up is the ANIMATION routine that reads paths for each of the objects you've defined, determines where shapes move to, then calls the PLOT routine. Your program only has to call the ANIMATION routine.

There are four animation types included in **Graphics Magician.** When you start out, you should only concern yourself with BLOCK mode. It's the simplest and fastest, and it's the type that you get automatically when you use the animation editor. The other types offer variations on speed, smoothness, collision detection, and use of backgrounds. Until you need changes in these, design your animations using BLOCK. You can always change later, very easily.

The four animation types are BLOCK, XDRAW, BLOCK with BACKGROUND, and BLOCK with BACKGROUND XDRAW. To understand some of the basic differences, you should understand basic animation cycles. Normally, to animate an object you draw it, update its coordinates, erase the old one, then draw the new, and repeat the sequence. In short, draw-update-erase-draw-update-, etc. The erase cycle takes just as long as the draw cycle in most instances, and it causes a moment when your shape is off the screen, causing some flicker in the animation.

The three BLOCK modes eliminate this erase cycle, only using draw-update-draw-update, and so on. The way this is done is that the shape erases itself when it's moved. First, you need to determine the maximum size move a shape will make (one dot, two dots,...). Then when the shape is created a border the size of this maximum move is left around the shape. The border is always the background color, usually black. Now, if a shape moves at most two dots, it has a 2-dot border as part of the shape. When it animates, this border, which is the color of the background, is plotted directly over any remnants of the shape from its previous position, effectively erasing itself! The shape remains on the screen (less flicker) and the erase cycle is eliminated (more speed).

Most things that come easily have some drawbacks, and BLOCK animation is no exception. First, if the shape is animating over a multicolored background, the border erases the background also. The solution turns out to be just as fast and rather easy at the expense of using 2 pages of hi-res memory, or 8,000 extra bytes. Both BLOCK with BACKGROUND modes preserve backgrounds, with additional options for helping with another problem.

This other problem is that collision detection is difficult with BLOCK mode, since the object is always colliding with itself (since it's never erased before the next plot, the new plot collides with the old). The only way to detect collisions is through the objects' coordinates, which in some cases can take time. The only surefire solution to this is XDRAW, which solves both the background and collision problems. Of course, it has its drawback: it requires the erase cycle all the time.

You should select the type of animation that best suits your needs. For most applications, if two hi-res pages are available for use, BLOCK with BACKGROUND should be the first choice, and BLOCK with BACKGROUND XDRAW an occasionally needed second choice. See the animation chart (on page 15) for a comparison of advantages and disadvantages.

BLOCK

Besides what's already been said about BLOCK animation, here are a few notes in summary:

BLOCK uses only one page of hi-res graphics, and that can be either page 1 or page 2.

BLOCK animation does not detect collisions and does not affect the collision table. It's fast and compact, but for many uses and requirements you will want to select BLOCK with BACKGROUND, which is just as fast, but gives more flexibility while also requiring more space.

XDRAW

XDRAW puts a shape on the screen by reversing whatever background is on the screen. The erase cycle simply re-reverses it, effectively restoring what was there before. Because it has an erase cycle, shapes that are going to animate using XDRAW do not need any border around them, hence can be stored a little more compactly.

When an XDRAW shape is put on a black background, it will appear just as it did in the shape editor. If it's put on a white background, it will appear in negative form. On any other color, some detail will be lost, but both shape and background will be preserved.

With XDRAW, all collisions register, since the objects never collide with themselves. The collision table will mark all collisions of objects with other objects AND with the background.

XDRAW only requires one page of hi-res, and that can be either page 1 or page 2.

The disadvantages of XDRAW are its slower speed and slight flicker because of the erase cycle.

BLOCK with BACKGROUND

Generally, BLOCK with BACKGROUND works the same as BLOCK. The difference is that it restores the background when a shape is moved. It does this by drawing only on page 1, while the background is stored on page 1 AND page 2. Before putting the shape on page 1, the shape is OR'd with page 2, so an imprint of the background is made on its border. Note that the border must be black, and that any black in the shape (or dots that are off) will also pick up the background.

BLOCK with BACKGROUND does pick up collisions of an object with the background, but not with other objects. It always draws on page 1, and always requires page 2 as an extra background screen. (Note that the page 2 background does not have to be the same as page 1. If it's different, the differences only appear when an object moves over them. This can be a tricky way to hide secret objects, or reveal a screen as it is moved over.)

There is an extra trick built into BLOCK with BACKGROUND that takes care of certain shortcomings of the BLOCK modes. One of these shortcomings not yet mentioned is that when BLOCK shapes overlap they tend to block each other out and flicker between the two shapes for the period that they are overlaid. The extra mode takes care of this in critical areas and helps with collision detection.

A table of flags is set in the two BLOCK with BACKGROUND modes that tell the animator that an object should be drawn normally, or that it should be animated on page 2! If it is also animated on the undisplayed page 2, it becomes a moving part of the background! What

does this all mean? First, since it becomes part of the background, collisions between it and other objects can be detected, since the other objects will show that the "background" was hit. Second, when it overlaps another shape, the other shape will preserve it as part of the background and reduce the flickering.

A few hints and cautions:

(1) Use only one or a few shapes on both pages. One key shape on page 2 will usually allow all important collisions to be detected.

(2) To maximize the correspondence between visual effect and what actually happens, your first object or objects (#0, 1,...) should be the ones moving on both pages. The animator moves the objects in numeric sequence, so the smallest delay is from objects 0 to N (where N is the last in the list), and the longest is usually from N to 0 (since other things go on in the program between calls to the animator). If the last object is the one to be drawn on both pages, collisions won't be picked up on its new position until the next call to the animator, at which point that object will move again. Visually, this could give the appearance of a delayed "hit", or of actually missing or picking up extra collisions.

(3) The object(s) drawn on page 2 are drawn using BLOCK mode. That means that they don't have any background to restore, and will wipe out any background they move over on page 2. BLOCK with BACKGROUND XDRAW solves this problem, albeit at the expense of a little speed.

BLOCK with BACKGROUND XDRAW

This mode is identical to BLOCK with BACKGROUND with the only exception being with objects drawn to page 2. In note (3) just above, the regular BLOCK with BACKGROUND uses BLOCK moves on page 2 and will wipe out any background that's there. BLOCK with BACKGROUND XDRAW uses XDRAW on page 2 and preserves the background. The expense at which this is gotten is that only the object(s) drawn on page 2 require an erase cycle, which won't add flicker, but will take a slight increase in time.

Other Things Affecting Speed

The three biggest factors in speed are the size of your shapes, the number of objects, and the code you use in between calls to the animator. Each time you move a shape, you are moving a block of memory. The more memory you move, the longer it takes, which explains the first two factors. The program code you use causes delays between animation calls. The example given in chapter 2 is about the fastest you can go from a Basic program. When you start adding a lot of code in between calls, your speed will be affected, so you want that code to be as fast and efficient as possible. Using machine language is one way to avoid excessive delays between calls, and there really is no substitute for machine language if you are writing something that's very involved with a lot of shapes, calculations, and comparisons and requires extremely fast speed. If you are using Basic, avoid large numbers of shapes, long calculations, IF statements, loops (other than the one to repeat the animation calls), and paddle reads as much as possible.

Table of Animation Types

	Speed	Graphics Storage	Collisions	Flicker	Background Destruction	Shapes
Block	Fast	8K page 1 or 2	None	None	Objects erase background	Must have background color border the size of maximum move
Xdraw	Medium	8K page 1 or 2	With background and all other objects	Some	No destruction	No border necessary
Block with Background	Fast	16K pages 1 and 2	With background and page 2 objects	None	No destruction on page 1; objects on page 2 erase background	Must have color #0, black, border the size of maximum move
Block with Background Xdraw	Fast to Medium, depending on number of page 2 objects	16K pages 1 and 2	With background and page 2 objects	None	No destruction	Must have color #0, black, border the size of maximum move

Chapter 4 - The Shape Editor

Background Color

The first things you see when you use the shape editor are a few questions that allow you to set some general information about your shape. First you are asked if you want to use the standard black background. If you will be using either version of BLOCK with BACKGROUND animation, always press Return for the default, yes. The only time you would choose "(**N**)o" is if you are using BLOCK or XDRAW and the background color for animation will be all one color, but not black. In other words, 99% of the time, take the standard.

If you do choose a background color, choose one of the standard Apple colors, 0-7. See Appendix A for information about colors on the Apple.

Load Previous Shape?

The second question is whether you want to load a previous shape. Choose "(**Y**)es" only if you have a shape that you've saved on disk and now want to modify, otherwise press Return.

Standard Border?

The third question is whether you want to use the standard border, which is 2. Usually press Return for yes and go on.

If you will be using XDRAW animation for this shape, you can use a border of 0, but if you later want to use that shape with another type of animation, you will have to come back to the shape editor and modify it to put on a border.

The default of 2 assumes a maximum move of 2 dots at a time in any direction (moves like "2 up and 2 to the right" qualify). A shape can move up to 7 dots in any direction in a single move, so the maximum border is 7. Color is preserved only on horizontal moves that are multiples of 2, however (0,2,4, and 6). You can give your shapes extra bursts of speed if you choose a larger border and use the longer movements, but the expense may be extra space taken by your shape (to accommodate the larger border).

You may also want to choose a border of zero and leave the border yourself (by not drawing in the left and right columns and top and bottom rows). Use this trick for shapes that will only move horizontally or only vertically. Leave the border where it's needed for erasing itself, but if a shape will only move horizontally, for example, the borders on the top and bottom are unneeded.

Width, Height, Shape Size

The last two questions asked before you get into the editor are the width and height of your shape (excluding borders). There is both a practical maximum and a real maximum. The practical maximum depends on the speed you desire, number of shapes, and storage space you have for your shape. The larger your shape, the slower it will be, and the more room it will take. The real maximum is 256 bytes, which roughly translates to a shape of about 36 by 42 dots, including borders, although various rectangles taller and narrower, or wider and shorter, can be used.

A way to compute the maximum, although the computer does it automatically and tells you if the shape is too large, is to first figure that your actual height in bytes, HB, is 2 times the border plus the shape height, or

HB = 2 * B + H

The width in bytes is a little more tricky. Given the border size, B, and the width in dots, W, the width in bytes, WB is:

WB = INT((W + 2 * B + 12) / 7)

This accommodates the extra byte taken by the shifting of the shapes horizontally. The result is that HB * WB must be 256 or less. Again, all these computations are done for you, but at some point you may want to know how to compute your own maximums.

One thing to note is that the formula for WB also tells you that certain increases in the total width don't affect the number of bytes used. If you consider the total width in dots as the shape width plus twice the border, table 4.1 shows the widths that fit into each number of bytes.

Table 4.1 - Byte width vs. Dot width

Bytes Wide	# Dots
1	1
2	2-8
3	9-15
4	16-22
5	23-29
6	30-36
7	37-43
8	44-50

Shape Editor Commands

When you get into the editor section, the commands listed in figure 4.1 should appear on the bottom of the screen.

Figure 4.1 - Shape Commands

```
I J K M Z X Q W C B F R S E A 1-7 O
WIDTH:          HEIGHT:          BORDER:

COLOR: WHITE  HBIT OFF  ALL SET
```

In the first row are the command letters you may use. The second row tells the width, height, and border that you selected, and the fourth row tells you the plotting color and some information about how it is formed. The third row will tell you the cursor's x,y coordinate in the shape.

Since you have already used the shape editor in the tutorial from chapter 2, here's a summary of the commands you've tried, along with a few that are similar:

Cursor Movement

I,J,K,M - cursor movement commands, up, left, right, and down, respectively, as they appear on the keyboard. Note that the cursor wraps around, so for example if you move your cursor off the top, it reappears on the bottom.

Plotting and Erasing

Z - plot one point in the current color.

Q - lock plotting on, so each time you move the cursor the new point is automatically plotted in the current color.

X - erase one point (turn the dot off).

W - lock erasing on, so that any point you move your cursor over is turned off.

Test Animate

(**A**)nimate lets you test animate your shape on the bottom of the screen. Speed is controlled by paddle or joystick. Return to editing mode by pressing any key.

Disk Options

(**O**)ptions selects disk options, for saving, loading, cataloging, clearing the shape and starting over, or returning to the menu.

Color

(**C**)olor selects a new plotting color. Your choices are 0-7, with 0=black, 1=green, 2=violet, 3=white, 4=black, 5=orange, 6=blue, and 7=white. Note that when selecting a new color the fourth line tells you the color and how the color is made, as described in appendix A.

An important note should be made about color. When you plot a dot "in the current color", you will have situations when a dot is not turned on at all. Either of the two whites will plot at any position. Orange and green plot only in odd columns of the screen (which may not necessarily be odd columns with respect to your shape), and blue and violet only plot in even screen columns. The effective screen resolution when working with blue, orange, green, or violet is 140 dots. With black and white it is 280 dots. That's why most programs that use graphics use a black background. Furthermore, of the four colors (excluding black and white), only orange and blue, or violet and green, can appear on the same horizontal line within a byte. Green cannot appear next to orange or blue, for example. Neither can violet. Vertically, however, you can have each line using any different color. For a full discussion of Apple colors, see appendix A.

Borders and Boundaries

(**B**)orders allows you to select new borders (the edge around your shape that lets it erase itself) and boundaries (the area in which you can draw your shape), either extending them to fit a larger shape than you expected, or bringing them in closer to save storage space for a smaller shape.

Flipping and Rotating

(**F**)lip allows you to flip your shape over left/right or up/down. This is handy if you have a shape that you want to be able to turn around and face a new direction, but don't want to redraw the whole thing by hand.

(**R**)otate lets you rotate your shape 90 degrees. Rotating four times brings it back to its original position. Rotating also exchanges the width and height of the boundaries.

Shifting

(**S**)hift lets you move your shapes up, down, left, or right within their boundaries. This is handy (1) if you underestimate or overestimate the size of your figure and need a little extra or less room, (2) if you decide you need a bigger or smaller border and need to move the shape a little to accommodate it, or (3) for a couple of tricks for still and vertical animation, discussed later.

Exchanging Colors

(**E**)xchange lets you try any combination of four color swaps (appropriately numbered 1-4). This won't let you make any one particular exchange of colors, but it will give a few options.

Internal Animation

Internal animation is when a figure not only moves around the screen, but it has features within the shape that move, such as legs moving in a walking motion, wings flapping, propellers rotating, eyes blinking, and so on. Pre-shifted shapes lend themselves easily to internal animation, since in reality for each shape you are creating seven frames. Those seven frames are what you see being drawn on the screen. By pressing any of the number keys **1-7,** you'll see that the orange bar over a frame is erased or redrawn. When the orange bar is on, that frame is being plotted on. When it's off, most commands you use will not affect that frame. By turning on and off the plotting on each of the individual frames, you can modify each slightly to create seven sequenced frames for internal animation.

The sequence of the frames depends on the number of dots the shape will move horizontally in each step. This will usually correspond to your border size, which will usually be two. The frame sequence for moves of two is shown below:

1 2 3 4

5 6 7

This corresponds to the 1-7 on the keyboard for turning the plotting on and off. Using the above numbers, the table below shows the sequencing for the various possible horizontal moves.

Horizontal Step	Frame Sequence
0	No internal animation
1	1,5,2,6,3,7,4
2	1,2,3,4,5,6,7
3	1,6,4,2,7,5,3
4	1,3,5,7,2,4,6
5	1,7,6,5,4,3,2
6	1,4,7,3,6,2,5
7	No internal animation

Note that with horizontal moves of 0 or 7, since the shape will always be in the same place with respect to byte boundaries, the same frame will be plotted and no internal animation will be seen. There are a couple tricks that can be used to overcome this, which will be discussed in the "Animation from your programs" section. Note also that 1 and 6, 2 and 5, and 3 and 4 are simply reverse patterns, so, for example, a shape created with frames that are sequenced for moves of 2 will also work with moves of 5 in many cases.

The best way to create a shape that animates internally is to first create the basic shape with all seven frames turned on. Then, once you are satisfied with the shape and colors, turn on and off frames so that you can alter each separately for the detail in the animation.

Disk Options

From the editor, if you press (**O**)ptions, you may then choose one of the following:

(**E**)ditor returns to the editor.

(**S**)ave the shape to disk.

(**L**)oad a previously saved shape from disk into the editor.

(**D**)isk catalog.

(**C**)lear the shape and start over.

(**Q**)uit the shape editor and return to the menu.

Chapter 5 - The Path Editor

The path editor lets you draw preset paths for your shapes. You can also program your objects to animate in paths controlled by a joystick or other means, but this is a way to make objects loop, or travel from one fixed point to another.

The paths you create are relative; that is, you define the shape of the path by drawing it, and no matter where your object starts it will follow that path. For example, one object can animate in a circle path at the upper left corner of the screen. Another object can animate in that same circle path, but at the lower right corner of the screen. The path is just a pre-defined set of moves, such as "up 2, right 4", relative to the current location of an object.

Remember that if your shape uses color other than black and white, you must use horizontal moves that are multiples of two for the color to remain correct. You can always move your shape vertically any number of units and retain the proper colors.

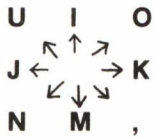
Background Screen

When you select "(**P**)ath editor" from the menu, you will first be asked if you want to load a background screen. If you have a screen picture that you want to animate over, and if it's saved in standard screen format (33 or 34 sector, with ".PIC" as a suffix; see appendix B for specifics), you can load it now and draw your path over the picture. Press "(**Y**)es" to load a screen, "(**N**)o" to skip, or "(**D**)isk catalog" to see what's on your data disk. If you choose to load a picture, just type the picture name, not the ".PIC". The suffix is added automatically by the program.

Starting Point

You will next be asked to move the flashing dot cursor to where you want to start your path. Usually the starting point isn't too important; since the path is relative, one starting point is as good as the next. However, if you are tracing over a background screen, or if your path is large, you may want to move to a corner of the screen so you have room to draw.

To move the cursor, use the keys shown below:



Notice that they are the same four keys for up, down, left, and right that are used in the shape editor, with four extra keys added for the diagonals. The directions correspond to their orientation on the keyboard.

When the cursor is where you want it, press "(**S**)tart". Now you are ready to actually draw the path.

Movement

The same eight keys used for positioning the cursor are used for drawing the path. The default step is 2 dots, so using "**K**" moves 2 dots to the right. Using "**N**" moves diagonally 2 dots down and 2 to the left. These eight keys all move AND register the move as part of the path.

There are also four extra cursor movement keys for fine tuning. On Apple II and II Plus, use **RETURN**, ← , → and "/" for up, left, right, and down (as they appear on the keyboard). On Apple //e, use the four arrow keys at the bottom right corner of the keyboard. These movement keys move in one dot increments in their directions, but do not register the moves until you press "**Z**" (the same key used for "plot" in the shape editor). Pressing any of the standard eight movement keys instead will move and register the standard movement.

You may also press "**Z**" before making any move to register a "no move", or pause.

The default standard move of 2 dots up/down and left/right can be easily changed by using the "**X**" and "**Y**" keys. "**X**" allows you to change the standard left/right move to 1-7 dots, and "**Y**" lets you change the standard up/down move to 1-7 dots.

Other Path Editor Commands

"(**D**)elete", or the **DELETE** key on the Apple //e, lets you delete the last move you made. You may use it repeatedly to delete several moves.

ESC allows you to toggle between full-screen graphics and graphics with text at the bottom. This is handy if your path goes down to the very bottom of the screen behind where the text lines are.

"(**S**)ave" lets you save your path to disk.

"(**C**)lear" lets you clear the path shown on the screen and start over. You are then given the chance to load a new background and start from the beginning.

"(**Q**)uit" lets you quit the path editor and go back to the menu. You are asked if you are sure that you are ready to quit, just in case you didn't save your path to disk.

Chapter 6 - The Animation Editor

The animation-editor is where you assemble all the parts of your animation by loading the shapes and paths that you plan to use, and assigning shapes, paths, and starting locations to the objects that will perform the final animation. You can create up to 32 different objects. Each can have a different shape and path, or you can have several using the same shape and/or the same path.

When you choose "(**A**)nimation editor" from the menu, you are given the choices of things you can do from that editor, as shown below:

(**L**)oad	(**N**)ames
(**O**)bject Create	(**H**)i-res clear
(**E**)dit Object	(**T**)ype of animation
(**A**)nimate	(**C**)lear
(**S**)ave	(**Q**)uit
(**D**)isk catalog	

You are also given the number of shapes and paths that have been loaded, the number of objects you've created, the animation type and page, and a "bottom address". Your animation file starts building in the top area of memory, just under Apple DOS, and works its way down (see figure 6.1). When you start, the bottom address is always 36608, but each time you load a shape or path, this number is moved down enough to make room. If you are using a large number of shapes and paths, you should watch the bottom address to see how much you are cutting into your program space, and more importantly, to see if you get close to overlapping a graphics page that you are using.

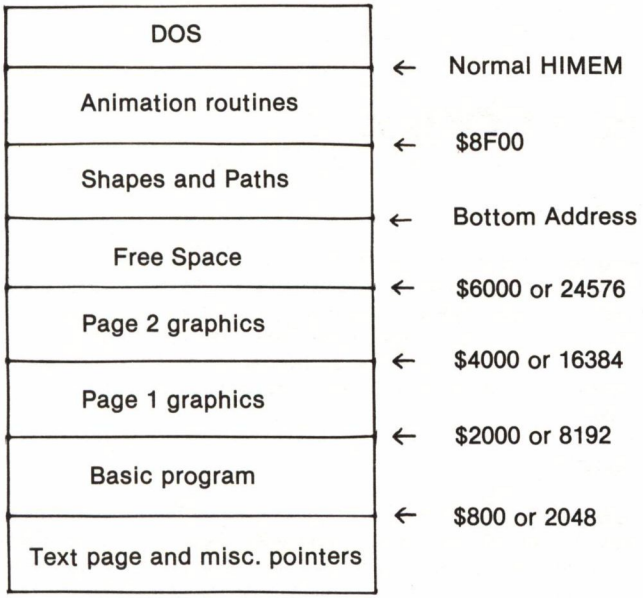


Figure 6.1 - Apple Memory Chart for Programs Using Animation

Loading

The first thing you should do is load at least one shape and one path. You can load as many as you want, but before you can create an object, you'll need at least one of each.

When you press "(**L**)oad", you'll then be asked whether you want a (**S**)hape, (**P**)ath, (**H**)i-res Screen, or previously-defined (**A**)nimation tables for editing or testing. You'll first want to load shapes and paths created with the shape and path editors. Press either "(**S**)hape" or "(**P**)ath", then the name of the shape or path that you used when you saved it. Note that if you cataloged the disk, your shapes have the suffix ".SSH" and the paths have the suffix ".PTH". You never have to type the suffix; it's just a way of preventing accidental loading of a shape named "CIRCLE", for example, as a path, which would have weird results. Note that when you load shapes and paths, the program numbers them in the order that you load them, starting at 0. Therefore, the first path you load is path #0, the second path you load is path #1, and so on. The numbering works the same for the shapes, so the first shape loaded is shape #0, the second is shape #1, etc.

You can also load a background screen that has been saved in standard 33 or 34 sector ".PIC" format (see Appendix B) for your animation to be drawn over. You should load the hi-res screen as the last step before testing your animation. The hi-res locations are used as a temporary loading area for shapes and paths, and as a result loading of shapes and paths leaves some garbage on the screen.

An existing animation file that has been saved with the animation editor can also be reloaded at this point for editing and testing. Any shapes or paths that you may have just loaded will be overwritten.

Creating an Object

Shape Number

To create an object for animation, press "(**O**)bject create". Just like the shape and path numbering, the first object is object #0. You will be prompted for a series of information, starting with shape number. Type in the number of the shape that you want this object to have. If you don't remember the numbers of the shapes, just press Return, and a list of the shape names and numbers will be given. Because of space constraints, only the first three letters of shape and path names are listed, so shapes like MOON and MOOSE will both be shown as "MOO".

Coordinates

Step 2 is to give a starting coordinate for your object. X=0 at the left edge of the screen, and increases to the right. The right edge of the screen is X=279, but you can enter a number for X anywhere in the range 0-1791. The actual "animation playfield" is 6.4 screens wide. It also "wraps around", meaning an object can animate in a straight line to the right, go off the screen, and it would eventually return on the left side.

The y-coordinate has Y=0 at the top of the screen, and increases as it moves down to Y=191 at the bottom. Y can have any value in the range 0-255, so the playfield is actually 1.3 screens tall. Combined, the entire playfield has over eight and a half screens for animation.

If your shape is in color, you should position it on an odd x-coordinate for it to have the same color as in the shape editor. Shifting it over one dot into an even coordinate will complement all the colors.

Path Assignment

Each object can have one to three paths in which to move. After moving in those paths, you can either repeat the sequence, stop the object and leave it on the screen, or stop the object and erase it.

Suppose you have three paths, one called CIRCLE, one called SQUARE, and one called TRIANGLE. You can do things like have an object move in a CIRCLE and keep repeating that circle. You can have another object move in a TRIANGLE, then disappear. Another can move in a SQUARE, then stop on the screen. Yet another can move in a SQUARE, then a CIRCLE, then a TRIANGLE, then repeat the sequence starting with the SQUARE again.

If a path isn't "closed" (that is, if the ending point isn't the same as the starting point) you can repeat the path right off the screen. The simplest case is a path that is a single move to the right. If that path is repeated, the object will just move until it goes off the right edge of the screen, later reappearing from the left edge (6.4 screens later). A circle that isn't complete, say nine-tenths of the arc, if repeated would cause the object to move in a spiral path until it goes off the screen... to reappear eventually from the other side.

To select the path or paths for an object, you will first be asked for PATH A. Type in the number of the path you want for the first in the sequence, or press Return to get a list of the paths and their numbers. After selecting PATH A, you'll be asked for PATH B, with the additional choices of "(**R**)epeat, (**D**)rawn at end, or (**E**)rased at end". Type the number of the path if you wish to select a second in the sequence, or press R, D, or E to end the sequence the way you want. If you selected a second path, you'll be asked for PATH C, with the same ending choices. If you select the third path, you'll then only be asked for the ending choice of R, D, or E.

Location in Path

The fourth step is to select the "path location". You do not have to start with the first move in a path; you may start at any step. Typically you'll type 0 for the first step. If you want to start an object 10 moves into the path, you can type 10. This allows you to set several objects on the same path, but out of sync with one another.

If you selected more than one path, you also have the choice of starting in path A, B, or C.

Test Animation

Now that you've got at least one object, you can see what your animation looks like. Usually you'll want to press "(**H**)i-res clear" first to clear any leftover garbage off the screen. In some instances you'll want to load a background screen instead. To try the animation, press "(**A**)nimate". The animation you defined will be shown on the screen. You can control the speed with a paddle or joystick, and when you're done watching, press any key to go back to the options. If you want to see it again from the start, just repeat the "(**H**)i-res clear/(**A**)nimate" sequence.

Edit Object

If, after testing your animation, you decide that an object needs something changed such as the starting location or a path, press "(**E**)dit object". Choose the number of the object that you wish to review or edit. All the information that you've given for that object will then be displayed on the screen, and you will be given the choice of changing the shape, the coordinates, the paths, or the starting path locations. By pressing the appropriate letter prompt, you are allowed to reenter any of that information. When done, press "(**R**)eturn to options".

Page 2 Flag - For Block with Background Only

If you had chosen one of the two "Block with Background" animation types (the default when you run the editor is "Block"), the edit options will display one more piece of information and an additional choice. A "Page 2 Flag" will be listed as having a value of 0 or 1. With the "Block with Background" animation types, remember, you have the option of having an object drawn on both page 1 AND page 2, thus becoming a temporary part of the background. If the "page 2 flag" is equal to 0, that means that the object will NOT be drawn on page 2 (this is the default). If the "page 2 flag" has a value of 1, that object will be drawn on page 2. During editing, pressing "(**F**)lag" will reverse the value of the "page 2 flag".

Changing Animation Types

You may change the type of animation from the default of "Block" to any of the other three types by pressing "(**T**)ype of animation", then selecting the animation type you desire. If you are using one disk drive, your master disk should be inserted before selecting (if it isn't already), since a file must be loaded from it. If you choose "Block" or "Xdraw", you also have a choice of animating on graphics page 1 or page 2. You may use this option at any time to switch an animation file to any other animation type.

Other Options

"(**S**)ave" saves all the information you used in creating your animation, along with the machine language routines necessary to perform it, into a pair of files on your data disk. The actual machine language file is given the suffix ".ANM". A text file which contains information necessary for re-editing and documenting your animation, but not for running it, is given the suffix ".ATX".

"(**N**)ames" lists the names of all the shapes and paths you've loaded, along with their numbers.

"(**D**)isk catalog" catalogs your data disk.

"(**C**)lear" allows you to clear your entire animation and start over.

"(**Q**)uit" returns to the menu, after checking if you really want to quit in case you haven't saved your animation.

Chapter 7 - The Animation Documenter

Pressing "(**D**)ocument animation" from the menu gets you to a utility that is not necessary to use, but sometimes comes in handy. The animation documenter will first of all allow you to print on the screen or a printer all the information about an animation file you've created. The information included will be useful when trying some of the tricks listed in chapters 9 and 10. (All of this information you could also get by yourself through other means, but this speeds up the process.) The documenter also has the feature of allowing you to delete unused shapes and paths from an animation file, thus freeing up extra space for your programs. Occasionally you will find that you've loaded a shape or path, only to realize later that it wasn't really needed for your animation. The delete feature in the documenter provides an easy way to free that wasted space.

In the documenter, after loading your animation file, you are given the choices of printing an animation report, deleting unused shapes or paths, saving a modified file (one that's been changed by deleting a shape or path), or quitting and going back to the menu.

Animation Report

When you choose "(**R**)eport", you are asked if you want it printed to the screen or a printer, and if a printer, in what slot # is the printer card (usually the printer card is put in slot 1). The generated report will then list the animation file name, the animation type and page, the bottom address of the animation file, the number of objects, paths, and shapes, the names and addresses of all the shapes and paths, and then all the information you entered for each object, including addresses at which your programs can access and modify that information. All addresses are given in decimal, following by hexadecimal in parentheses. When the shape and path names and addresses are listed, the report will print "UNUSED" next to any that are not used by any of your objects.

A printer is recommended for this report, and you will find the report useful to have on hand when you start trying some of the programming tricks in the following chapters.

Deleting Shapes and Paths

If any of the shapes or paths are flagged as unused in the animation report, you may delete them by pressing "(**D**)elete shapes or paths". You will be prompted to choose a shape or path and its number. If that one is indeed unused, it will be deleted. You will not be allowed to delete shapes or paths that are used in the animation. If you are deleting more than one shape or path, be aware that each time you delete, the numbers are shifted down. For example, if you delete shape 2, shapes 0 and 1 retain their numbers, but shape 3 now becomes 2, shape 4 becomes 3, and so on. To overcome confusion and speed up the process, delete the highest numbered shape or path first, then follow in reverse sequence. Deleting a shape with a high number does not affect the numbering of those below it.

Saving the Modified Animation File

Remember that if you delete any shapes or paths you will have to save the changed animation file, or next time you load it they will still be there.

Chapter 8 - Animation from your Programs: Necessities

Once your animation file is saved to a data disk, you can write your own programs to access and control the animation. The way the machine language animation routine works is that each time you call the routine it performs one step in the animation; that is, each object is moved one step along its path.

The simplest program for running an animation is given in listing 8.1. Line 10 sets the hi-res graphics mode to page 1; the poke clears the bottom four text lines. Line 20 loads the animation routines from disk. Line 30 calls the animation routine to perform one step of the animation, and line 40 loops back up to 30 so that the program repeatedly performs single steps of animation.

```
10 HGR : POKE - 16302,0
20 PRINT CHR$ (4);"BLOAD FIRST.ANM"
30 CALL 36864
40 GOTO 30
```

Listing 8.1 - Simple Animation from Basic

Between what occurs in lines 30 and 40 is where you have control over what happens to the animation. Each time the animation routine is called and a step is performed, all the animation tables are checked and updated. Before the routine is called again you can change shapes and paths, activate and deactivate objects, check for collisions, and anything else you may want to do in your program.

The general program design for using animation will be:

1) Set-up and initialize all variables, graphics screens, load the animation file, etc.

2) Call the animation routine.

3) Check and change parameters.

4) Go back to step 2.

In listing 8.1, step 3 is bypassed. For most programs, though, this step will be the most critical. The commands that are put in step 3 are executed between calls to the animation, so the longer they take, the slower the animation will be. If there is a lot to do here, there may be good reason to use machine language for this step. No matter what appears at step 3, it should be written as efficiently as possible.

As you can see in listing 8.1, there are few commands needed to run your animation. In describing the ones that are necessary, though, it is important to first know which type of animation you are using: Block, Xdraw, Block with Background, or Block with Background Xdraw. The most frequently used are Block and Block with Background, and unless you changed types in the animation editor yourself, your file is set up to use Block.

Set-up

First, you want to set the hi-res display, as in line 10 of listing 8.1, and load your animation file, as in line 20. If you use any string variables, you should also use a statement such as:

25 HIMEM: (bottom address)

where "(bottom address)" should be given as the number that is the bottom address listed in the animation editor (and documenter) for your animation file. This protects the area of memory in which the animation routines are located. Using this command will never hurt, so it's recommended whenever in doubt.

Animation Calls - Block and Block with Background

For Block, or Block with Background, each time you want one step of the animation performed, use CALL 36864 (or JSR $9000 in machine language). This calls the main animation routine. It will perform one animation step, update the tables, and return to your program.

Animation Calls - Xdraw and Block with Background Xdraw

For Xdraw, or Block with Background Xdraw, it is necessary to call a "Firstdraw" routine before the loop. (In listing 8.1, we'd put it at line 28 or thereabouts). Since Xdraw actually does an erase-draw cycle, it is necessary to have the objects on the screen to erase the first time. "Firstdraw" simply does a draw cycle, so that the normal Xdraw routine can be used after it. (The astute will note that with Xdraw, an erase is a repeated draw; i.e. of two Xdraw operations, the first will draw and the second will erase. The Firstdraw routine does one Xdraw operation to put the object on the screen. The normal Xdraw routine does one Xdraw operation to erase the object, gets the new coordinates, then does another Xdraw operation to draw the object again.)

When using the Xdraw routine, Firstdraw is called using CALL 37073 (or JSR $90D1 from machine language).

When using Block with Background Xdraw, use CALL 37117 (or JSR $90FD) for the Firstdraw routine.

For either, use the same animation call as with the regular Block modes: CALL 36864 (or JSR $9000).

Chapter 9 - Animation from your Programs: Options and Tables

Following is a list of all the animation tables that are used and kept by the animation routines. Most are not used frequently, and it's recommended that you skim them briefly and go on to the next chapter, which gives examples of using most of them and will allow you to refer back. This is one of those technical sections that has some information that one may never use, but is handy to have around anyway.

Each table is listed with its addresses in hexadecimal and decimal. Quick calculations are shown for finding exact locations for any object, and the value N is used in those calculations to designate object number (possible values 0-31).

Hi and Lo addressing

In the tables, references are made to the Hi and Lo bytes of an address. An address in the computer takes two bytes of storage, and since each byte can hold values ranging from 0 to 255, the addresses are split modulo 256. It's not necessary to understand if you simply use the Basic commands:

```
Hi = INT(A/256)
Lo = A - Hi*256
```

where A is the address.

Now if this two-byte address is stored for reference in two consecutive locations, there are two orders in which to put them: Lo,Hi or Hi,Lo. Oddly enough, most addresses are stored in Lo,Hi format in machine language. **Graphics Magician** also usually uses Lo,Hi format, except where noted otherwise.

Shape Index $9380-$93FF (decimal 37760-37887)

This table contains the actual memory addresses of each of the shapes. Each address is two bytes long, in Lo,Hi format. There is room for 64 addresses, hence the shapes may be numbered 0-63. If S is the shape number, its address may be found as:

```
Lo=PEEK (37760+2*S)

Hi=PEEK (37761+2*S)

Address=Hi *256+Lo
```

You may change this address anytime during animation to change the shape of an object being animated (see Object List, also) or to allow more than 64 different shapes to be used in a program.

Object List $9400-$941F (decimal 37888-37919)

This is a list of shape numbers for objects 0-31. Each entry is one byte long, so to find the shape number for object N, look in location 37888+N. This number may be changed with Block and Block with Background to allow an object to change shape in mid-animation. (The new shape should be close to the same size, or larger, so that the old shape is fully erased.)

More importantly, this list is the one to which the animation is keyed. This list gives the sequence of movements of the objects for each animation step (object 0 goes first, then 1, and so on), and flags the last object to animate. When the animator finds a shape number 255 ($FF), it assumes that the last object has been moved, so it stops and returns to your program. Example: if object 0 has shape #5, object 1 has shape #2, and object 2 has shape #255, the animator concludes that only two objects are being animated. The 255 flags the end of the list.

Also, objects may be temporarily deactivated using this list. If the shape number is greater than 127, but less than 255, the animator skips that object and continues with the next one in the list. So if you want object N to "sit still" for a moment, you can take the current shape number, add 128 to it, and store it in the object list, deactivating the object (if you don't want to take the computer time to do the calculation, you can just put a 128 there). To reactivate an object, just subtract 128 from the number in the object list to get the original shape number back. In machine language, this operation is equivalent to setting and clearing the high-bit.

Page 2 Flags $9420-$943F (decimal 37920-37951)

Used in Block with Background and Block with Background Xdraw, these locations hold values that tell whether an object will appear on page 1 only, or animate on both page 1 and page 2. For object N, the location of its Page 2 Flag is 37920+N. If an object is to be drawn on page 1 only, this location holds the number 17 (hexadecimal $11). Objects drawn on page 1 and page 2 will have the number 145 (hex $91) in this location. Any other numbers will give strange results. The reason: for speed, these numbers correspond to actual instruction codes that are inserted into the machine language routines.

Path Locations

The path location is described in three sub-tables; the current path (a number 0,1, or 2 — corresponding to path letter A, B, or C in the Path List), and the actual address of the next step in the object's path, split into Hi and Lo address form in two tables for speed.

Path Letter $9440-$945F (decimal 37952-37983)

The path letter corresponds to the Path List table, described later, and simply tells if the object is on its first, second, or third path (A,B,or C, given by 0,1, or 2). When the path sequence is finished and there is no repeat used, this value is set greater than 127 (negative in machine language), thus deactivating the object. You may use this location as an alternate to the Object List for deactivating objects. Object N's current path number is at 37952+N.

Path Address Lo $9460-$947F
(decimal 37984-38015)
and
Path Address Hi $9480-$949F
(decimal 38016-38047)

This is the pointer to where the next path value for this object is stored in memory. Upon start of the animation, the two-byte address pointer usually points to the start of the first path being used. After each animation call, this pointer is incremented to the next address, until a zero (end-of-path) value is found, at which point the Path List is checked for the next action. You may use this pointer to trick the animator into thinking a path is done by pointing this location to any byte that has a value of zero. To compute the address, A, of the next path value for object N, use:

```
PEEK(38016+N) * 256 + PEEK(37984+N)
```

Object Locations

This is also a 3-part list that contains three values for each object that give its current x,y coordinate. The coordinate refers to the upper-left corner of the shape. The first two bytes for each give the x, and the third byte gives the y. The y location is a value 0-255, with 0-191 on the screen, and 192-255 as a wraparound area. For speed, the x location is stored in byte/bit format. The first location tells in which horizontal byte the object is located, and the second tells the bit, 0-6. Although only 40 bytes across are displayed on the screen, the byte number may range from 0 to 255, with 0-39 displayed, and 40-255 as a wraparound area. The object location may be changed directly by your program, but if it is not currently in a wraparound area (i.e., if it is visible) you must also use an erase routine to remove the object at the old location.
The bit number is also the number that tells the PLOT routine which frame of the shape to use.

X-Byte $94A0-$94BF
(decimal 38048-38079)
and
X-Bit $94C0-$94DF
(decimal 38080-38111)

To find the x location of object N, use:

```
X=PEEK(38048+N)*7+PEEK (38080+N)
```

Y-Location $94E0-$94FF
(decimal 38112-38143)

Find the y location of object N using PEEK (38112+N).

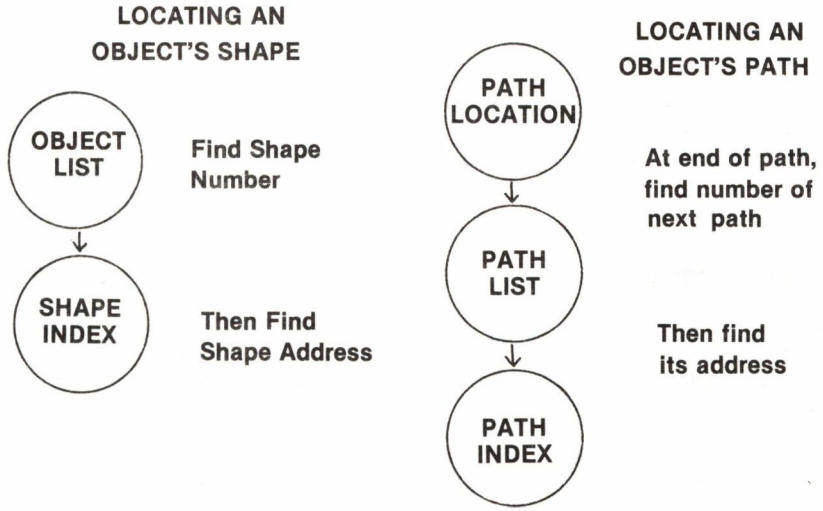
Path Lists $9500-$957F (decimal 38144-38271)

The Path List is four bytes long for each object. It contains the numbers in the Path Index of each object's three paths (A, B, and C from the animation editor), and a repeat flag. The first byte of this information for object N may be found at 38144+N*4.

The first byte (byte 0) is the number of path A; its address may be found in the Path Index. If there is no path B, byte 1 is the repeat flag. If there is a path B, byte 1 contains its number corresponding to the Path Index. Then, if there is no path C, byte 2 contains the repeat flag. If there is a path C, byte 2 holds its number, and byte 3 holds the repeat flag.

The repeat flag has a value of 255 ($FF) if the sequence is to be repeated. Its value is 254 ($FE) if the sequence should end without erasing the shape. Or, if the value is 253 ($FD), the sequence ends with the shape erased.

The path numbers in the Path List are only used when the Path Location list hits the end of a path. When that occurs, the animator then looks to the Path List to see what to do next. It will either go on to the next path or follow the instructions of the repeat flag.



Path Index $9580-$95DF (decimal 38272-38367)

Like the Shape Index, this table gives a two-byte address in lo, hi format for each path. There is room for 48 paths, numbered 0-47. The Path List refers here for finding the address of a new path to put into the Path Locations table. To find the address of path number P, use:

```
Hi=PEEK (38273+2*P)

Lo=PEEK (38272+2*P)

Address=Hi*256+Lo
```

Collision Table $95EO-$95FF (decimal 38368-38399)

The collision table contains one byte for each object. This byte contains a zero unless the corresponding object incurred a collision during the previous animation step, in which case it is set to any non-zero value. The collision flag for object N is found at 38368+N.

The collision flag is used as follows for each animation type:

Block - unused

Xdraw - detects all collisions with other objects and the background.

Block with Background - page 1 objects pick up all collisions with the background and page 2 objects. Page 2 objects do not detect collisions.

Block with Background Xdraw - Page 1 objects pick up all collisions with the background and page 2 objects. Page 2 objects pick up collisions with the background and other page 2 objects.

A collision is detected when an "on" dot in a shape is put in a screen position where an "on" dot already exists, either from another shape or from the background. Note that because of the way Apple colors are made, some colored shapes will never collide with one another. A blue shape will never collide with an orange shape, for example, since blue is made up of even dots, and orange is made of odd dots. See appendix A for color details.

Path Format

When controlling the movement of an object with a joystick, keyboard, or even some formula, it becomes necessary to know how each move is described to the animator. The next chapter will contain some examples of controlling animation through input devices. This section will describe the actual format of the path.

Each move in a path takes one byte. A path with 100 moves in it will take 101 bytes of storage. The last byte contains the value zero, which flags the end of a path. Whenever a zero value is reached in a path, the animator looks at the Path Lists to see what to do next, whether to go to another path, repeat the path sequence, or stop.

The eight bits in each path byte are divided into two sets of four (see figure 9.1). The first four bits give the horizontal x movement and the second four bits give the vertical y movement. Each group of four bits can contain the values -7 to +7. The leftmost bit in each group is a sign bit. If it's set (1), the value is negative, if it's off (0), the value is positive. The remaining three bits give the binary values for 0 to 7. See figure 9.2 for the binary/decimal equivalences. The following formula in Basic will give you the proper path value for a move of DX, DY, where DX is the x-move and DY is the y-move:

```
PV = 128 * (DX<0) + 16 * ABS(DX) + 8 * (DY<0) + ABS(DY) : IF PV=0 THEN PV=8
```

Note that in Basic you can use expressions like (DX<0) that give a value of 1 if true and a value of 0 if false.

Another way to figure the path value is to look at the chart in figure 9.2, find the actual values for the moves you want for DX or DY, and use 16*DX+DY.

A positive value for DX will move the object to the right; a negative value for DX will move it to the left. A positive value for DY will move the object downward; a negative value will move it up.

Note that since a zero flags the end of a path, we cannot use it to designate "no move". Instead, as shown in figure 9.2, you can use any combination that gives you a negative zero, which is still no move. The path values 8, 128, and 136 all give a "no move", or pause, since they are all combinations of zeroes and negative zeroes.

Figure 9.1 - How a Path Value Appears in a Byte

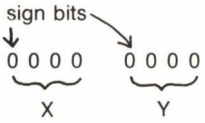


Figure 9.2 - Path Values in Binary

Binary	Move Described	Actual Value To Use
0 0 0 0	0	0
0 0 0 1	1	1
0 0 1 0	2	2
0 0 1 1	3	3
0 1 0 0	4	4
0 1 0 1	5	5
0 1 1 0	6	6
0 1 1 1	7	7
1 0 0 0	-0	8
1 0 0 1	-1	9
1 0 1 0	-2	10
1 0 1 1	-3	11
1 1 0 0	-4	12
1 1 0 1	-5	13
1 1 1 0	-6	14
1 1 1 1	-7	15

Shape Format

Each shape contains 16 bytes of leading information, followed by the actual bit maps of the seven frames. The first 14 bytes are two-byte offset pointers to the individual frames. The offset to frame 0 is in bytes 0 and 1, the offset to frame 1 is in bytes 2 and 3, and so on. The offsets are stored in Hi,Lo format. To find the start of the actual bit map for a frame, you would add the starting address of the shape to the offset for that frame.

Bytes 14 and 15 are the width and height of the shape, in bytes. These numbers tell how to interpret the bit map. If W is the width and H is the height, the first W bytes of the bit map give the top horizontal line of the frame. The next W bytes give the second line, and so on for H lines.

The Nitty-Gritty Plot Routines

The animator is actually two routines. A main routine cycles through all the objects, computing new locations by the information in the object location and path tables, and calls the plot routine to put each object on the screen (or to erase it). The plot routine simply puts an object on the screen or takes it off. Given the shape number, it knows which frame to put in the location sent by the animator by the bit number it is given for the location.

It is possible, and in special circumstances useful, to use the plot routine without going through the animator. To plot, not animate, a single shape at a given x,y coordinate, you must do the following for any of the four animation/plot types:

1) Put the x coordinate in locations 0 and 1; the x-byte in 0, and the x-bit in 1.
   (e.g. POKE 0, byte: POKE 1, bit)

2) Put the y coordinate in location 2.

3) Put the shape number (not object number) in location 3.

4) CALL 36608 or JSR $8F00

The Block routine does not return a collision value. Xdraw returns a collision value in location 11 ($B), and Block with Background and Block with Background Xdraw return collision values in location 254 ($FE).

Calling Erase Routines

You may also call erase routines, which are part of the plot routines. For all types repeat steps 1-3 above. For Xdraw, repeat step 4 also, since the draw and erase functions are the same. For the other three types, use CALL 36751, or JSR $8F8F. The Block erase will put a black background over the shape. The Block with Background erase routines will erase page 1 shapes by placing the background over them. Shapes on page 2 will not be affected.

Erasing on Page 2

To erase a page 2 shape in regular Block with Background, you can just plot a blank shape over the one you want erased. To erase a page 2 shape in Block with Background Xdraw requires a little programming patchwork. In the Blockback/Xdraw code, locations $9031-$9049 contain a code swapping routine that temporarily changes the Blockback plot routine to an Xdraw plot routine, calls it, then changes it back. By setting up bytes 0-3, then extracting that code and putting it into your own, you can erase an Xdrawn shape on page 2. If you want that shape redrawn at some point without calling Firstdraw, you'll have to repeat the procedure. If you simply want to swap shapes, use the same steps, except after the JSR $8F00 in the extracted code insert the set-up of locations 0-3 again for the new shape and another JSR $8F00.

Machine Language Memory and JSR Chart for Animation Routines

	Block	Xdraw	Block with Back.	Block with Xdraw
Memory Usage	$0-$A $8F00-8FDE $9000-90C9 $9180-95DF	$0-$B $8F00-8F80 $9000-90E8 $9180-95FF	$0-$C, $FE $8F00-8FE7 $9000-90E0 $9180-95FF	$0-$C, $FE $8F00-$8FFB $9000-9114 $9180-95FF
Call Location	$9000	$9000	$9000	$9000
FirstDraw Call	none	$90D1	none	$90FD
Plot Routine	$8F00	$8F00	$8F00	$8F00
Erase Routine	$8F8F	$8F00	$8F8F	$8F8F

Note: Normally the space from $9600 and up is taken by DOS and its buffers. Using a MAXFILES 1 command will free two buffers and let you use the space $9600-$9AA5.

Chapter 10 - Animation from your Programs: Samples and Tricks

This chapter has several examples of how to use the various controls over the animation. There are always other ways, and of course there are tricks and techniques that we haven't even discovered yet, but the examples in this chapter should answer a lot of basic questions about how some of the tables can be used. All the sample programs are in Basic, but the same techniques can also be used from machine language.

Controlling Paths with Joystick or Keyboard

The first trick in having an object controlled by a joystick is to create a path that is a single move, we'll call it ONE MOVE. The move can be in any direction, since we'll constantly alter it from our program. In the animation editor, when creating the object that you want to control independently, assign it the path ONE MOVE, followed by a repeat. (If you will want more than one object controlled by a player, and each to move independently — for example with a two-player game with each person holding one game paddle — you will have to load in ONE MOVE twice as two different paths and assign one to each of the independent objects.) Note which object number is the one you will control, then save the animation file. There is a sample of this type of animation file called "JOYSTICK.ANM" on the data side of your **Graphics Magician** disk.

Joystick Example

Listing 10.1 shows a sample program for controlling an object with a joystick. In this case we used our sample JOYSTICK.ANM file, which has the controllable object as #0. Line 10 loads the animation file, and line 20 sets the hi-res graphics mode. (Line 20 also clears page 2, since the animation type of JOYSTICK.ANM is Block with Background. Though we don't use the background in this example, we will with the same animation file in listing 10.4.) Line 30 just says that the object number that we'll control is #0. Line 40 looks in the Path List and Path Index tables to find the address of the first path for object #0. This will be the starting address of our ONE MOVE path in memory.

Now is where the loop begins. First, line 50 reads the two paddles (a joystick is equivalent to two paddles, one for the vertical and one for the horizontal). Line 60 then does a computation to find a path value for the paddle readings we found. Note again the computations using something like (X<80) as a value. A comparison expression like that has a value of 1 if true, 0 if false. Of the two expressions (Y<80) and (Y>160), only one can be true, so the "Y" part of the computation, (Y<80)*10+(Y>160)*2, will result in a value of either 0, 2, or 10, the path equivalences of 0, +2, or -2 for the y change (see figure 9.2 again). Similarly, the "X" part of the computation gives either a +2, 0, or -2 for the x change. Lastly, if it turns out to be a zero move, the path value is changed to 8, which is a negative zero and won't flag the end of the path. Line 70 pokes the path value into the actual path address that we found and calls the animate routine. Line 80 loops back up to start the paddle-read process all over.

```
10 PRINT CHR$ (4);"BLOAD JOYSTICK.ANM"
20 HGR2 : HGR : POKE - 16302,0
30 N = 0
40 T = PEEK (38144 + N * 4):P = PEEK (38272 + T *
   2) + PEEK (38273 + T * 2) * 256
50 X = PDL (0):Y = PDL (1)
60 M = (X < 80) * 160 + (X > 160) * 32 + (Y < 80) *
   10 + (Y > 160) * 2: IF M = 0 THEN M = 8
70 POKE P,M: CALL 36864
80 GOTO 50
```

Listing 10.1 - Joystick Control of Animation

Keyboard Example

Listing 10.2 shows a similar example using keyboard for input instead of a joystick. Lines 1-40 are the same, except for the HIMEM command, which has been added since we're using a string variable here. We'll set the controls so that the space bar is stop, the A and Z keys are up and down, and the right and left arrow keys (ASCII character values 21 and 8, respectively) are right and left.

Line 45 sets the initial move to "stop". Line 50 checks for a keypress. The PEEK used can be found in your Applesoft manual. If the value returned is greater than 127, a key has been pressed. Otherwise one hasn't, and the program skips the key read and path change. All we do differently now is check which key has been pressed, and if it's one that gives us a new direction, we assign that direction value to the path.

```
5 HIMEM: 36055
10 PRINT CHR$ (4);"BLOAD JOYSTICK.ANM"
20 HGR2 : HGR : POKE - 16302,0
30 N = 0
40 T = PEEK (38144 + N * 4):P = PEEK (38272 + T *
   2) + PEEK (38273 + T * 2) * 256
45 POKE P,8
50 IF PEEK ( - 16384) < 128 THEN 120
60 GET A$: IF A$ = " " THEN M = 8: GOTO 110
70 IF A$ = "A" THEN M = 10: GOTO 110
80 IF A$ = "Z" THEN M = 2: GOTO 110
90 IF A$ = CHR$ (8) THEN M = 160: GOTO 110
100 IF A$ = CHR$ (21) THEN M = 32: GOTO 110
110 POKE P,M
120 CALL 36864: GOTO 50
```

Listing 10.2 - Animation Control from Keyboard

Collisions

Listing 10.3 is an example of how to check the collision table. Our sample, COLLIDE.ANM, uses Xdraw animation, so any collision between the two objects should be detected. Object 0 has a path of ONE MOVE, and we'll control it with a joystick. Object 1 has a path we chose at random, and we'll let it go its own merry way.

Lines 10-70 are exactly the same as listing 10.1 for joystick control, except (1) we are loading a different animation file in line 10, (2) the POKE that clears the text lines has been omitted in line 20, and (3) there is a call to the Firstdraw routine in line 45, since we're using Xdraw animation now. The collision table starts at location 38368, so the collision flag for object 0 is at 38368 and the collision flag for object 1 is at 38369. Lines 80 and 90 have been added, which print the values of the collision flags each time through the loop, and beep and stop if a collision occurs. Any keypress resumes the animation.

When running this example, note that sequence determines if one or both collision flags are turned on. One object may move with no collision, then the other object may move on top of it, causing a collision to register with the second. Likewise, one object may move on top of the other, flagging a collision, but in its turn the second may move away enough so that in its new position there is no collision.

```
10 PRINT CHR$ (4);"BLOAD COLLIDE.ANM"
20 HGR2 : HGR
30 N = 0
40 T = PEEK (38144 + N * 4):P = PEEK (38272 + T *
   2) + PEEK (38273 + T * 2) * 256
45 CALL 37073
50 X = PDL (0):Y = PDL (1)
60 M = (X < 80) * 160 + (X > 160) * 32 + (Y < 80) *
   10 + (Y > 160) * 2: IF M = 0 THEN M = 8
70 POKE P,M: CALL 36864
80 PRINT PEEK (38368), PEEK (38369)
90 IF PEEK (38368) OR PEEK (38369) THEN PRINT
   CHR$ (7): GET A$
100 GOTO 50
```

Listing 10.3 - Checking the Collision Table

Listing 10.4 shows an example of collision detection with a background. We're using the JOYSTICK.ANM file again, since it is done in Block with Background mode. Again, lines 10-70 are the same as in listing 10.1, with the addition of two lines that load a background picture, MAZE.PIC, on page 1 (line 22) and page 2 of graphics (line 24). This time lines 80 and 90 only check for collisions of object 0 with the background.

```
10 PRINT CHR$ (4);"BLOAD JOYSTICK.ANM"
20 HGR2 : HGR
22 PRINT CHR$ (4);"BLOAD MAZE.PIC,A$2000"
24 PRINT CHR$ (4);"BLOAD MAZE.PIC,A$4000"
30 N = 0
40 T = PEEK (38144 + N * 4):P = PEEK (38272 + T *
   2) + PEEK (38273 + T * 2) * 256
50 X = PDL (0):Y = PDL (1)
60 M = (X < 80) * 160 + (X > 160) * 32 + (Y < 80) *
   10 + (Y > 160) * 2: IF M = 0 THEN M = 8
70 POKE P,M: CALL 36864
80 PRINT PEEK (38368)
90 IF PEEK (38368) THEN PRINT CHR$ (7): GET A$
100 GOTO 50
```

Listing 10.4 - Collisions with Background

Using the Object List

The next example uses one of the demo files on your disk, LETTERS.ANM. This animation file contains 27 shapes: the letters of the alphabet plus a space. Each of 28 objects is assigned to the same rectangular path, although each has a different starting point in that path. The result is that the letters will move in single file, since each is slightly ahead of the last in the path. The starting step in the path and the x,y coordinates had to be computed in advance by giving each object a starting location and counting the path moves between each. (This animation file, as well as all the others in this chapter, can be loaded into the Animation Editor or Animation Documenter and examined closely, or even be changed.)

This example lets the user type in a message, then takes each letter of that message and assigns the shape of that letter to the next object in sequence using the Object List. The important new lines are those from 60 to 90. Line 65, in particular, takes the ASCII code of the next letter in sequence and subtracts 65. The ASCII code for the letter A is 65, with each of the other alphabetic characters following in order, so the letter A returns a value of 0, B becomes 1, and so on, which are the equivalents of their shape numbers in LETTERS.ANM. Lines 70 and 75 poke the shape number into the object list. If the character is not a letter, it pokes a 26, which is the shape number for a space (in this particular file). Line 90 fills up all the remaining 28 objects with spaces.

```
5 HIMEM: 26336
10 PRINT CHR$ (4);"BLOAD LETTERS.ANM"
15 HGR2
20 TEXT : HOME
30 PRINT "TYPE SOMETHING 28 CHARACTERS OR LESS.":
   INPUT A$
40 IF LEN (A$) > 28 THEN 30
50 HGR : POKE - 16302,0
60 FOR I = 1 TO LEN (A$)
65 A = ASC ( MID$ (A$,I,1)) - 65
70 IF A < 0 OR A > 25 THEN POKE 37888 + I - 1,26
   : GOTO 80
75 POKE 37888 + I - 1,A
80 NEXT I
90 FOR J = I TO 28: POKE 37888 + J - 1,26: NEXT :
   POKE 37916,255
100 CALL 36864: IF PEEK ( - 16384) < 127 THEN 10
   0
110 GET A$: GOTO 20
```

Listing 10.5 - Putting Shapes in the Object List

Of note here is that when the letters are moving vertically, they are put in columns that are multiples of seven (0,7,14,21...). This is because the letters were designed with frame 1 as the full letter, with frames 2-7 being the various animated stages. Frame 1 only appears in columns that are multiples of 7. Figure 10.1 is a table showing the columns in which each frame appears.

Figure 10.1 - Frame/Column Chart

Frame	Column
1	Multiples of 7
2	Multiples of 7 plus 2
3	Multiples of 7 plus 4
4	Multiples of 7 plus 6
5	Multiples of 7 plus 1
6	Multiples of 7 plus 3
7	Multiples of 7 plus 5

Deactivating Objects

The Object list can also be used for activating and deactivating objects. Listing 10.6 is a crude game of sorts, in which you control one object with a joystick while the others move around randomly until you collide with them, at which time they freeze. The animation file COLLGAME.ANM used Xdraw animation with six objects. Object 0 is controlled with the joystick, and objects 1-5 move in random paths. Each object was given a path of ONE MOVE, although ONE MOVE was loaded six times, as paths 0-5, so that each object could have an independent path. Object 0 has path 0, object 1 has path 1, object 2 has path 2, and so on.

The program sets HIMEM to the bottom address of the animation file, then loads the animation file COLLGAME. Another file, COLLISION DETECT, is then loaded. This is an extra little machine language routine that scans the collision table quickly and tells you whether something has collided. It's very short (15 bytes) and resides in a low part of memory (768, or hex $300). When you call the routine at 769, it scans the collision table until it finds a non-zero value, then puts that object number in 768, and returns. If there was no collision, the number 255 is put in location 768. As is, the routine scans through all 32 objects in reverse order (31 to 0). If you have fewer objects, you can poke the number of the last object in location 770. In our case, with objects 0-5, we'll poke 770 with a 5 (see line 50).

To find the locations of each object's path, we used an array, P, and assigned P(0) the location of the path for object 0, P(1) the location of the path for object 1, etc. The computation is the same as the one we used before, in line 40. Line 50 sets the number of objects for the collision detect routine and calls Firstdraw for the Xdraw animation.

Lines 60-75 select random paths for each of the objects 1-5. The statement INT(RND (1)*255)+1 gives a random number from 1 to 255... all valid path moves.

Lines 80-150 loop 15 times before selecting new random paths for each object; otherwise if the path were selected at random for each move they'd look like they were doing the jitterbug. The paddle read and joystick move commands from lines 90 to 110 should look familiar by now. Line 120 checks the collision detect routine. If the collision registered on your object (#0), or if there was no collision (C=255), the loop continues at line 150. If there was a collision with an object 1-5, line 130 deactivates that object by poking 128 into the object list, and line 135 clears the collision flag for that object. The collision detect routine is then checked again, in case you collided with two objects at once.

```
5 HIMEM: 36018
10 PRINT CHR$ (4);"BLOAD COLLGAME.ANM"
15 PRINT CHR$ (4);"BLOAD COLLISION DETECT"
20 HGR : POKE - 16302,0
30 FOR N = 0 TO 5
40 T = PEEK (38144 + N * 4):P(N) = PEEK (38272 +
   T * 2) + PEEK (38273 + T * 2) * 256
45 NEXT
50 POKE 770,5: CALL 37073
60 FOR I = 1 TO 5
70 POKE P(I), INT ( RND (1) * 255) + 1
```

```
75 NEXT I
80 FOR I = 1 TO 15
90 X = PDL (0):Y = PDL (1)
100 M = (Y < 80) * 14 + (Y > 160) * 6 + (X < 80) *
   224 + (X > 150) * 96: IF M = 0 THEN M = 8
110 POKE P(0),M: CALL 36864
120 CALL 769:C = PEEK (768): IF C = 0 OR C = 255
    THEN 150
130 POKE 37888 + C,254
135 POKE 38368 + C,0: PRINT CHR$ (7)
140 GOTO 120
150 NEXT I
160 GOTO 60
```

Listing 10.6 - Deactivating Objects

It's not much of a "game". Frequently the objects will scatter off screen and you'll have to wait for them to wander back onto the visible part of the playfield. They can also collide with themselves and put themselves out of commission. It should give you a starting point for playing with the various options you can use with the information tables, though.

Switching Shapes Midstream

You can also use the object list to change the shape of an object during animation. The only cautions are that if you are using one of the three Block modes, the new shape should be the same size or larger than the old shape, and if you are using an Xdraw mode, you must call the erase routine separately first.

Listing 10.7 has a short example using the LETTERS.ANM file. After each fifty moves, the letter is changed by poking the next shape number into the object list. Note that all objects after #0 are deactivated by poking object #1 with a 255.

```
10 HIMEM: 26336
20 PRINT CHR$ (4);"BLOAD LETTERS.ANM"
30 HGR2
40 POKE 37889,255
50 HGR : POKE - 16302,0
60 FOR L = 0 TO 25
70 POKE 37888,L
80 FOR I = 1 TO 50
90 CALL 36864
100 NEXT I: NEXT L
110 GOTO 60
```

Listing 10.7 - Changing Shapes

Still and Vertical Animation

One limitation you may discover occasionally is that internal animation only occurs when an object is moving horizontally. There is a trick to getting around that if you design a special shape and a special path. When your object moves vertically or stops in one place, you'll want to put that shape number into the object list and use your special path.

Step 1 - Design your shape in the shape editor. You don't need any borders on the left or right, but if you will be moving vertically, leave borders at the top and bottom. Create the internal animation in the sequence A) 1,2,3,4,5,6,7 or in the sequence B) 1,5,2,6,3,7,4. Which you choose doesn't matter, as long as your path corresponds later.

Step 2 - Don't save your shape yet. Using (**Control-S**)hift, a special variation of the shift option that shifts over borders, and turning the frames on and off appropriately, do the following:

Move frame 4 six dots to the left.
Move frame 7 five dots to the left.
Move frame 3 four dots to the left.
Move frame 6 three dots to the left.
Move frame 2 two dots to the left.
Move frame 5 one dot to the left.

Some of your frames will be outside the boundaries. That's okay.

Step 3 - Use the "(**B**)orders" option and decrease the width by subtracting 6. This may save some space by eliminating one byte in width.

Step 4 - Save the shape.

Step 5 - In the path editor, create a path with one of the following sequences for x moves, depending on whether you chose sequence A or B for internal animation of your shape:

Sequence A	Sequence B
Right 2	Right 1
Right 2	Right 1
Right 2	Right 1
Left 5	Right 1
Right 2	Right 1
Right 2	Right 1
Left 5	Left 6

Along with these moves for x, you can mix them with any moves you want with y for vertical animation (i.e., "Right 2" could just as well be "Right 2 and Up 4" in one move).

Step 6 - In the animation editor, or whenever you want to put this in effect, use the shape and path you just created in a column that is a multiple of seven and you'll get static or vertical animation.

Part Two - The Picture System

Chapter 11 - Drawing Pictures

The picture drawing system is designed to let you create screen pictures that take a minimal amount of storage space. Pictures always take 8K, approximately 8000 bytes, of storage to display on one of the hi-res pages. However there are ways that they can take considerably less storage on disk. About 12 standard 8K pictures can fit on one side of a floppy disk. With **The Graphics Magician**, you can easily fit fifty to well over a hundred pictures on a single side of a disk.

Standard 8K pictures are stored as the values in the 8192 bytes that make up the hi-res screen. With **The Graphics Magician**, instead of storing the results of your drawing as a screen image, the moves that you make in creating your drawing are stored. The moves for most drawings can be in hundreds of bytes instead of thousands. We call these "sequential pictures", since the sequence is remembered instead of the actual picture.

The effect this has is that the computer "remembers" what you do as you draw. Later, when you want to view that picture again, the computer simply reconstructs your moves, very quickly. If you've played any graphic adventure games, you probably will recognize what this looks like; the picture redraws very rapidly before your eyes. What you see recreated are the moves that the artist made while drawing the picture the first time. Most adventure games use this technique (many of them done with **The Graphics Magician**), since they demand that large numbers of pictures fit on a disk. There are also many educational products that use **Graphics Magician** this way, since they also require a large number of graphic images.

There are four types of "moves" that you, the artist, can make. You may draw a line, fill an enclosed area with color, plot a computer "brush", or type a letter over your picture. In addition, you may choose from a palette of over 100 color mixes, and select one of eight different brushes, ranging from a small, precision size to a large airbrush effect.

Using the Picture Editor

From the menu, select the version of the picture editor for your input device. Most people will use the "(**J**)oystick/paddle version", which also works with other devices that give joystick-type input, such as a trackball.

You'll first be asked if you want to "(**L**)oad a previous picture" for further editing, start a "(**N**)ew picture", load a "(**B**)ackground" so that you may make an overlay, or see a "(**D**)isk catalog". The first time through, choose "(**N**)ew picture".

Now you'll see a white screen, with a few text lines on the bottom, as in figure 11.1. If you move your input device around, for discussion's sake we'll assume with a joystick, you'll see a small cursor in the shape of a crosshair move around the picture. (You may also see a second stationary crosshair). This is what you control for drawing your picture. (If the cursor does not seem to correspond at all to your joystick movement, you may have a non-standard joystick in which the axes are reversed. Pressing "J" will switch them for you.)

Joystick Version Cursor Control

The Apple screen has a resolution of 280 by 192 dots. A joystick has a resolution of 256 by 256. To compensate for the lower horizontal resolution, the screen actually has two zones. Most of the screen lies in both. Position your cursor at the left edge. Now move it slowly to the right. It will move smoothly until you get very near the right edge of the screen, where it will jump all the way to the right edge. Move it back, and you'll see that it moves smoothly over that area now, until you get very close to the left edge, at which point it will jump back to the left edge. Generally, the joystick positions you on either the left 90% of the screen or the right 90% of the screen. The only time you see that jump is if you move all the way from one edge to the other; hence you don't see it when doing detail work, only when moving a long range across the screen.

```
(T)EXT  (Z)ERO SP  ESC  R  D  E  S  I  Q
LINE      FCOLOR: 0      LCOLOR: 0
picture instructions listed here
BYTES USED: 1          X:____    Y:____
```

Figure 11.1 - Picture Editor Command Lines

Line Mode

When you first start, you are in line mode. Pressing button 1 on your joystick (or the RETURN key for versions with only one switch on the input device, such as an Apple Graphics Tablet) will move the stationary cursor to the current loction of your moving cursor. This is a "Start Line" command. The stationary cursor is the starting point of the next line. Pressing button 0 on the joystick (or pressing down the pen on an Apple Graphics Tablet) draws a line from the stationary cursor (Start Point) to the movable cursor (Ending Point). The ending point also becomes the new starting point. Move your cursor around and try the effects of buttons 0 and 1.

How Long is Your Picture?

When playing with line mode, note that three things happen on the bottom of the screen. The x,y position keeps changing as you move the cursor, and each time you press a button, the second to last line tells you what you just did ("Start Line at ---", or "Draw Line to ---"), and the bottom line tells you how many bytes you've used for your picture. Each "start line" or "draw line" command takes 3 bytes.

Deleting Steps

The first nice thing to learn is that pressing "(**D**)elete", or the **DELETE** key on an Apple //e, will delete the last step. If you make a mistake, it's easy to back up as many steps as you want and try again. Note that each time you delete a step, you see how the redraw option works. The program remembers what you've done to that point and recreates everything except the step you deleted.

If you have a lot of steps to delete, you can also press **Control-D** (hold down the CONTROL key and press D). You'll be asked how many steps you want to delete. The program still deletes the steps one at a time, but it cycles through automatically the number of times you specify.

Fine Cursor Control

Locate the cursor near a point that you want to hit exactly. Pressing "(**Z**)ero" zeroes in on that area for precision control. Move your joystick to the extremes; it doesn't go very far. It will move in a 40-dot by 40-dot area around the point you zeroed in on. Pressing "(**Z**)ero" again puts you back into normal mode.

Want even finer control? "(**Z**)ero" toggles between "zeroed in" and normal modes. Pressing **Control-Z** controls how precise the zero mode will be. Find a point and "(**Z**)ero" in on it. Move the joystick around to see the 40-dot range. Now press **Control-Z** and move the joystick around. It moves over a very tiny area, only 10 dots by 10 dots. **Control-Z** always toggles between the 10-dot and 40-dot modes. If you are using normal mode you won't notice the difference until you go to zero mode.

Selection Page

Okay, now for some fun. Press the SPACE BAR. A graphic screen appears that shows your choices for modes across the top (line, fill, or brushes) and pictures of the eight brushes, a strip on the left showing the eight possible line colors, and a palette showing the 108 possible fill colors. Your joystick controls an arrow that moves around this screen. The "(**Z**)ero" option works on this screen also, so if you're in zero mode, you may have to press "(**Z**)ero" again to free the arrow for full-screen movement.

An arrow near the top should point toward the line option, meaning that you are in line mode. Pointing your arrow at the fill option or one of the eight brushes and pressing button 0 on the joystick will select that option and put you in a different mode.

A second arrow points to the top black color on the line color strip on the left. This means that all your lines will be drawn in black. You can point your arrow at any of the other seven colors, and pressing button 0 will select that for line color on new lines. The eight line colors correspond to the standard eight Apple colors, as described in Appendix A. Either black color is recommended for all lines, since they provide the best borders for filling.

The third arrow points to the upper-left corner of the palette, to white. That is the current color used for filling and for brushes. Moving your arrow to any other color and pressing button 0 selects that color for future fills and brushes. Appendix A also contains notes on arrangement of colors on the palette, and reasons why the three whites you see on the palette are actually all different.

Select fill mode and a fill color other than white, then press the SPACE BAR. The SPACE BAR switches between the drawing screen and the selection screen. While viewing the selection screen, you may change any of the options that you want, or you may just check the options and colors and press SPACE BAR again to get back to the drawing screen.

Fill Mode

When in fill mode, you can fill any enclosed white area with the current fill color by positioning your cursor inside the area you want to fill and pressing button 0. The area should be white, with borders in black lines or the edge of the screen.

The fill routine is designed to be as fast as possible, so that pictures that are reconstructed in a finished program will appear very quickly. Most irregular areas will require two or three fill commands to fill the entire area, since with speed comes some compromise with completeness of fill. The fill routine used works the following way:

1) Scan directly up from the selected point until a border is found, then move down one line.

2) Fill to the left and right borders.

3) Average the left and right borders to find the midpoint, and move down one line from there.

4) Check to see if the point moved down to is a border. If not, go back to step 2.

The basic thing to remember is step 1. It means that the best place to position your cursor is anywhere directly below the uppermost point in the area to fill. Using this one trick will minimize the number of fill commands necessary for filling any area.

Brushes

From the selection page, choose any of the eight brushes by pointing to it and pressing button 0. Go back to the drawing page and you'll see that your cursor is now in the shape of the brush that you selected. Each time you press button 0, the brush will be plotted with the color you selected.

The brushes give you a very large amount of control over detail, shading, and effects that cannot be achieved with "line and fill" coloring-book graphics. You will probably want to start most pictures by laying down a background with lines, then adding most of the colors with fills, and finally adding the detail touches with brushes.

If you've used our other graphics programs, you'll notice immediately that there is no "brush up/brush down" selection. Each time you press button 0 the brush plots just once. If you want to move across an area, you have to keep pressing and releasing the button. While strange seeming at first, remember that the computer is remembering your moves. If the brush were constantly down, it would have to remember each and every point that you move over, wasting a lot of memory very quickly.

Other Quick and Easy Options, Including Saving your Picture

While drawing, you have these following other choices available at all times. The letter commands are listed at the bottom of the screen.

ESC - the **ESC**ape key switches between graphics and text display (the normal mode, with the four text lines at the bottom of the screen), and full-screen graphics mode. It has no effect on the picture itself, since the graphics area under the text is always available and is not overwritten by the text.

"(**R**)edraw" will reconstruct the picture as it would be seen from a program. This is handy if you are trying for some animation (explained later), and most people admit that it's fun just to see what you have drawn redone at blinding speed by the computer.

"(**S**)ave" allows you to save your picture in the special compact format created by **Graphics Magician**. To view this picture later, you will have to load it back into the picture editor, or follow the instructions for using the PICDRAW routine in chapter 13.

The "(**S**)ave" command only saves the displayed portion of the picture, which should be remembered when in edit mode, described below. This allows a way to extract parts of pictures, if desired.

"(**I**)mage" saves a standard screen image of the picture displayed on the screen. This is saved in the standard 34-sector ".PIC" format, and should be used if you want to save the background for use with the animation system in this package or if you want to edit your picture with some other graphics utility.

Note that there is NO way to edit a previously saved picture in a 33/34 sector ".PIC" format with this picture editor. Pictures created with other graphics editors cannot be converted to the special compact format of **Graphics Magician**. Since it is the moves that are saved, the pictures must be created with **The Graphics Magician** picture editor to have this compact format.

"(**Q**)uit" lets you return to **The Graphics Magician** menu or start a new picture. You are asked to verify that you really want to quit, in case you haven't saved the picture you are working on.

Adding Text

You can add text to your picture at any time by positioning your cursor where you want the text to start and pressing "(**T**)ext". Whatever you type now will be overlaid onto your picture at that point. The text mode takes control of the cursor, and you have the following options, which are listed in the text area at the bottom of the screen:

Control-L allows you to modify the keyboard input of an Apple II or Apple II plus to give lower case letters. Notice that the second text line on the screen says "KEYBOARD: STANDARD". Each time you press **Control-L** (by holding down the CONTROL key while pressing L), you switch between "standard" and "lower" keyboard. When in "lower", all the letter keys give lower case letters. Other keys get slightly confused. Pressing **Control-L** again gives the standard keyboard back.

With an Apple //e, when you enter text mode you can just release the "CAPS LOCK" key for upper and lower case input. When leaving text mode, you should re-engage the caps lock.

Control-R and **Control-N** let you choose between reversed letters and normal letters in color. Reversed letters flip whatever colors are on the screen. Normal letters plot in the current brush/fill color. The current type used is given on the second text line at the bottom of the screen. Which type is better depends on the background color and letter color desired. Some combinations are not feasible on the Apple, and experimentation is the best way to judge. The sharpest resolution comes with a combination black and white for the letters and background. Also, the first black and first white on the palette are a little more fuzzy than the other two due to the makeup of Apple colors.

ESC - Escape toggles to full-screen mode and back, just as before.

Control-D, **Back arrow**, or, on the Apple //e, **DELETE**, allow you to backspace and delete the last character.

RETURN, or reaching the end of a line, returns you to normal drawing with joystick control and in line mode. If you reach the end of a line, you'll hear a beep.

You may also reenter text mode at the last text cursor position by pressing **Control-T** while in normal drawing mode. Pressing "(**T**)ext" by itself always puts you in text mode at the position of the joystick cursor.

Edit Mode

Since the picture editor saves pictures as a set of moves, it is possible to go back and edit those moves, much like a computer program itself. An edit mode allows you to single-step forward and backward through a set of picture commands, displaying what the picture looks like at each point, and allowing you to delete or add moves at any time. If you decide later that you don't like a color, find where you set that color and set a different one. If a house needs a few extra lines before the color is filled in, backstep to before the color was added and put in the lines. Easy!

Pressing "(**E**)dit" while in normal drawing mode will clear the screen to white and position you in your "picture program" at the first move. Each time you press the **right arrow**, the next move in sequence will be displayed in words at the bottom of the screen and performed to the picture. Pressing the **back arrow** will back up one step. Your entire picture is still stored in memory, and pressing "(**R**)edraw" will bring it all back, but the edit mode allows you to add to or delete things that you did early in drawing your picture and see, step-by-step, how it was constructed.

You can tell if you are in edit mode by the inverse arrows (< >) at the right side of the command line at the bottom of the screen. All drawing commands remain usable, and anything you draw while in edit mode will be inserted into your picture. You can also use the "**D**" or **DELETE** keys just as before to remove commands.

It is also possible to "back into" edit mode from the end of a picture by using the **back arrow**. Stepping forward through a picture is faster, but if you have a long picture and want to edit one of the last moves, this makes it easier.

"(**R**)edraw" will always let you out of edit mode by redrawing the entire picture stored in memory. You will also get out of edit mode if you single-step through the entire picture and reach the end.

"(**S**)ave", when used in edit mode, will only save the part of the picture that is displayed. This is a convenient way to save only the first half of a picture, for instance. If you want to save the whole picture, you should use "(**R**)edraw" to get out of edit mode.

While single-stepping in edit mode, you can speed up by using either ">" (a greater-than sign, which looks a little like a right arrow), or **TAB** (on the Apple //e) to tab forward 10 steps at a time.

A "Fill" Anomaly

There is a seldom-occurring instance where the fill routine may seem to refuse to fill an area. Here are the circumstances, and the solution to the situation.

Figure 11.2 shows part of a drawing where the top and left side have been filled by a color. Since this is printed matter the lines appear in inverse, but assume that the solid fill lines are white. A few colors in the palette are blended so that alternate lines are solid white, and those are the ones that can cause this occurrence.

Now suppose you try to fill the area on the right. The fill routine will search upward and find the bottom white line of the area already filled, assume it is also to be filled, and proceed. Since that line stretches much further to the left, when the routine averages the endpoints and attempts to step down to the next line to fill, it will step down on the left side, which already is filled!

The solution is to break the white line extending across the top of the fill area by placing a single black dot atop the triangle in that line. The fill routine will now average endpoints that fall into the area that really needs to be filled, and it will work properly.

(Such are the prices one pays for speed and compactness of code. In fairness to the truths of programming which say that anything can be done better, if anyone devises a more complete fill routine that is as fast and space efficient, let us know. Our routine is 286 bytes long. It also shares y-lookup and color tables, and 82 bytes of common code for setting initial x,y address position and color registers, with the rest of the routines. Speed comparisons can easily be made by timing against the existing code.)

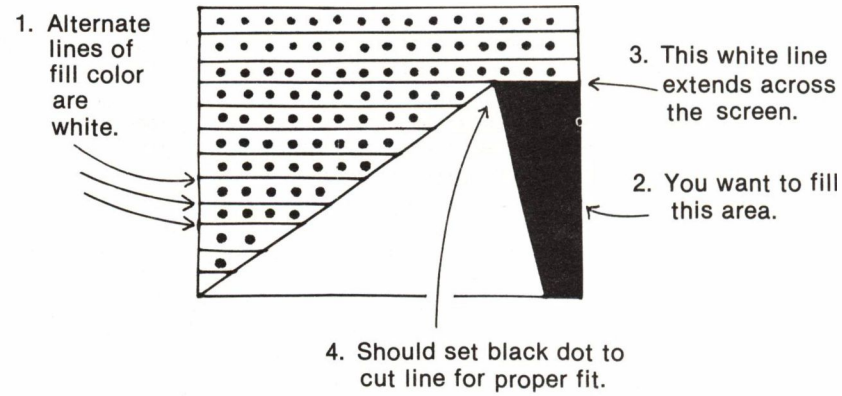


Figure 11.2 - Fill Anomaly

Chapter 12 - Tricks with Pictures

Objects

One of the features of many programs that require compact pictures is the ability to move objects from picture to picture, or to draw a picture with an object sometimes appearing, sometimes not. The obvious example is an adventure game, where something will appear in a picture, you take it with you, and thus it should no longer appear in the picture. We'll call this type of thing an "object". (No relation to animation objects in Part One of this manual.)

Objects with **The Graphics Magician** picture system are actually the same as pictures. You create them in generally the same way as you would any other picture. However, in your own program, when you use the picture-reconstructing "PICDRAW" routine, you tell it to draw that "picture" as an overlay at a certain coordinate. The picture thus becomes an object in the picture previously displayed.

The only requirement for objects that is not necessary for pictures is that the first command is a "Start Line" command. This is really a dummy "Start Line" command and will later be interpreted as a "Start Object at" command for object positioning. If you actually need a "Start Line" command, you must use a second one.

Since objects are usually drawn over other pictures, you should be careful about what types of commands and colors you use. Try to avoid colors and lines that would cause a "bleeding" effect when you place the object over the picture. Refer back to appendix A for a discussion of how this effect occurs and how to avoid it.

With objects you should also be careful about when you use a fill command, if it is used at all. Since the fill command requires a white background, you must be sure a white background is in place before you can fill. When you cannot be sure that the background picture will provide a white fill area, if you use a fill command you should first "white out" the area with one of the larger brushes.

For detail, most objects are done primarily with brushes and some careful use of lines. Some prefer to set down a white background with brushes first, no matter what, so that all the drawing commands can be used freely. It's usually a matter of style.

Creating your Object over a Background

To make it easier to choose colors and see how your object will look, you can draw it directly over a background picture. The first choice when running the picture editor was whether to start a new picture, edit an existing one, or load a background. If you load a background, you may draw your object directly on top of it. The background does not become part of your object, nor does it affect it in any way. Each time the screen is cleared, the background is displayed in place of the white screen that normally is shown.

Animation with Pictures

One of the effects discovered after **The Graphics Magician** was completed was that of animation created with the picture editor. Suppose you draw an entire picture, and in the picture is a man. Your commands in creating that picture are saved, so each time you view the picture, you'll see it redrawn. Now, suppose that once the picture is complete, you draw more right on top of what you finished. For example, draw the man's eyes closed, then go back and draw them open again. When you press "(**R**)edraw", you'll see the man being drawn to completion, then his eyes will blink! You can extend it out further and keep drawing over and over the original picture and make all kinds of things in the room "animate".

This effect can be accomplished by drawing continuously on top of a picture, or it can be done by drawing a sequence of objects over your picture. The latter method has the advantage of allowing more control in timing, even tying it to user responses in your program. To do it with objects, you might load in the background of the man, then draw one object of his closed eyes, and another of his eyes open again. Or you might make one object his closed eyes quickly overlaid by his open eyes, since it's an immediate progression. In your program you'd draw the background picture, then whenever you wanted his eyes to blink, you would draw your blinking eye object(s). You could have it controlled by time, or have it happen every time someone touched a key. The flexibility is yours!

Chapter 13 - Using Pictures in your Programs

When you created your pictures in the picture editor, what was saved was your moves in drawing the picture. To display it, you need some way to tell the computer to recreate those moves. The machine language routine called PICDRAW does just that.

Put the PICDRAW routine on your disk

First, you must move the PICDRAW routine from the **Graphics Magician** disk to your program and data disk. To do so, quit the **Graphics Magician** program so that you get the "]" prompt, then use the following steps:

1) Type MAXFILES 1, then press RETURN.

2) Put in your Graphics Magician disk, type BLOAD PICDRAWH, and press RETURN. PICDRAWH is a version of PICDRAW located high up in memory. The other version, PICDRAWL, is located toward the bottom.

3) Put in your disk and type:

BSAVE PICDRAWH, A$8D00, L$C00

then press RETURN.

Using PICDRAW

After the PICDRAW routine and your picture are on the same disk, you can write a program like the one is listing 13.1.

```
2 PRINT CHR$ (4);"MAXFILES1"
5 HIMEM: 32768
10 PRINT CHR$ (4);"BLOAD PICDRAWH"
20 PRINT CHR$ (4);"BLOAD PNAME.SPC,A32768"
30 HGR
40 POKE 0,0: POKE 1,128
50 CALL 36096
```

Listing 13.1 - using PICDRAW from Basic

The HIMEM command makes sure that the PICDRAW routine and the area in which you loaded your picture will not be disturbed. The PICDRAW routine starts at location 36096 ($8D00), and we'll load the picture commands at 32768 ($8000). Line 10 loads the PICDRAWH routine that you put on your disk, and line 20 loads in your picture. Note that ".SPC" was automatically added to the name when saved by the picture editor, and that the ",A32768" tells the program to load the picture commands starting at location 32768.

Line 30 sets the high-resolution graphics display, then line 40 pokes the starting address of the picture commands into locations 0 and 1. The address is poked in Lo/Hi format, which is computed as follows:

Hi = address divided by 256, with remainder left off.

Lo = remainder after address is divided by 256.

or mathematically, as:

HI = INT(A/256)

LO = A - HI*256

where A is the address. For listing 13.1, 32768 is the value we used for the address, and the numbers poked in line 40 were computed using that value with the above formulas. Listing 13.2 shows how we could have alternately used "A" and the formulas to have the program do the computation.

Line 50 contains the call to the PICDRAW routine at 36096 (or from machine language use JSR $8D00) that will cause the picture to be redrawn.

```
1 PRINT CHR$ (4);"MAXFILES1"
5 HIMEM: 32768
10 PRINT CHR$ (4);"BLOAD PICDRAWH"
20 PRINT CHR$ (4);"BLOAD PNAME.SPC,A32768"
30 HGR
40 A = 32768:HI = INT (A / 256):LO = A - HI * 256: POKE
   0,LO: POKE 1,HI
50 CALL 36096
```

Listing 13.2 - Using PICDRAW and Computing "Pokes"

To summarize what's necessary, all you have to do is load the PICDRAW routine once, then for each picture you want to display, load that picture file and use 2 pokes and a call.

Putting an Object over a Picture

The same technique used for redrawing a picture is used to redraw an object. Once the background picture is shown, load the object picture, and use the same two pokes to tell the location of the object commands. Then you have a choice between 2 calls.

CALL 36099 (or JSR $8D03) will draw the object over the picture at the same location in which the object was originally created. It is the same as drawing a picture, except the screen is not cleared beforehand.

CALL 36102 (or JSR $8D06) will allow you to specify an x,y coordinate at which to have the object drawn. The x,y coordinate should first be put in locations 36105-36107 as follows:

POKE 36105, X-(X>255)*256

POKE 36106, X>255

POKE 36107, Y

Technically, "xlo" is put in 36105 ($8D09), "xhi" is put in 36106 ($8D0A), and "y" is put in 36107 ($8D0B).

If the second call is used, the x,y coordinate must be such that the entire object will be shown on the picture. If any part of the object goes off the edges, it will not work properly.

Listing 13.3 can be used as a continuation of listing 13.1, and shows an example of drawing an object over a picture using alternate coordinates. If it should be drawn in its original location, line 90 could be omitted and the call in line 100 could be changed to CALL 36099.

```
2 PRINT CHR$ (4);"MAXFILES1"
5 HIMEM: 32768
10 PRINT CHR$ (4);"BLOAD PICDRAWH"
20 PRINT CHR$ (4);"BLOAD PNAME.SPC,A32768"
30 HGR
40 POKE 0,0: POKE 1,128
50 CALL 36096
60 PRINT CHR$ (4);"BLOAD ONAME.SPC,A32768"
70 POKE 0,0: POKE 1,128
80 X = 200:Y = 53
90 POKE 36105,X - (X > 255) * 256: POKE 36106,X >
   255: POKE 36107,Y
100 CALL 36102
```

Listing 13.3 - Adding an Object

Changing Graphics Pages

PICDRAW will put the picture on whichever graphics page is currently being drawn to, depending on whether you used an HGR or HGR2. To make PICDRAW put the picture on

the page that is not being seen, you can poke a value into location 230 ($E6). If it is a 32, the drawing will go to page 1 of graphics, if the value is 64, the drawing will go to page 2. Changing this location affects all drawing of graphics, even those using Applesoft commands from your program.

Technical Trivia

Before a picture is drawn, you must put the starting address of the picture file into locations 0 and 1, in Lo/Hi format. After the picture is drawn, locations 0 and 1 contain the first address after the picture commands.

Loading Groups of Pictures into One File

You can put several pictures in one long file using the following technique. This was used in the demo program in **Graphics Magician** so that all the pictures would load at once and the disk would not have to be accessed between each picture.

First, you must find the length of each picture you plan to use. This can be done by noting the "BYTES USED" in the picture editor for that picture, or by using the binary transfer utility described in chapter 15. Suppose the results were as in figure 13.1.

Figure 13.1 - Sample Picture Lengths

Name	Length	Load At
House.SPC	**1254**	**16384**
Tree.SPC	**879**	**17638**
Moose.SPC	**2318**	**18517**

You next need to choose a starting location for your group of pictures. For maximum use, start just at the end of the first hi-res graphics page, at location 16384 ($4000). This leaves room for 19,712 bytes of pictures. Now, load each of your pictures sequentially in memory. For the first, you'd use:

```
BLOAD HOUSE.SPC,A16384
```

which loads the picture at 16384. For the second, add the length of "House" to 16384 to find the next available space. 16384+1254=17638, so we load the second picture, in this example, at 17638:

```
BLOAD TREE.SPC,A17638
```

Remember the location at which you load each picture, since that's the address you must poke into locations 0 and 1 before redrawing the picture.

The third file is loaded at 17638+879=18517:

```
BLOAD MOOSE.SPC,A18517
```

Now, compute the total length by adding the lengths of all the pictures together (1254+879+2318=4451), and save the entire file with:

```
BSAVE PICTURE GROUP,A16384,L4451
```

substituting the name you want for PICTURE GROUP, and the appropriate starting address and length for your pictures.

Using a Group of Sequential Pictures

Listings 13.4 and 13.5 give examples of using picture groups in your programs. Listing 13.4 cycles through the pictures in order, waiting for a keypress between changes. Listing 13.5 uses the addresses we noted to allow you to select which picture is shown, and when. Pressing 1, 2, or 3 draws the appropriate picture. The programs in 13.4 and 13.5 should have the names and addresses changed to match the ones you used before you try them (the ones used are okay for the sample file already on disk), and lines 40 in listing 13.4 and 25, 30, and 40 in listing 13.5 should have the "3" changed to the number of pictures you use.

```
2 PRINT CHR$ (4);"MAXFILES1"
5 HIMEM: 8192
10 PRINT CHR$ (4);"BLOAD PICDRAWH"
15 HGR
20 PRINT CHR$ (4);"BLOAD PICTURE GROUP,A16384"
25 REM LINE 25 CONTAINS 16384 PRE-COMPUTED IN L
   O-HI FORMAT
30 POKE 0,0: POKE 1,64
40 FOR I = 1 TO 3
50 CALL 36096: GET A$
60 NEXT
```

Listing 13.4 - Cycling through a Group of Sequential Pictures

```
2 PRINT CHR$ (4);"MAXFILES1"
5 HIMEM: 8192
10 PRINT CHR$ (4);"BLOAD PICDRAWH"
15 HGR
20 PRINT CHR$ (4);"BLOAD PICTURE GROUP,A16384"
25 DIM L(3): REM L CONTAINS THE LOCATIONS OF EAC
   H PICTURE
30 FOR I = 1 TO 3: READ L(I): NEXT : DATA 16384,
   16765,16983
40 GET A$: IF A$ < "1" OR A$ > "3" THEN 40
45 V = L( VAL (A$))
50 POKE 0,V - INT (V / 256) * 256: POKE 1, INT (
   V / 256)
55 CALL 36096
60 GOTO 40
```

Listing 13.5 - Choosing from a Group of Sequential Pictures

Interesting to note in listing 13.4 is that the starting addresses do not have to be poked in each time if the pictures are shown sequentially. Since after each picture is drawn locations 0 and 1 point to the next location after the picture, they already point to the next picture in the list!

SINGLE STEPPING

If you need to slow down the PICDRAW routine, or if you want to draw a picture in parts, there is a way to make the routine single-step through the picture, returning to your program after each instruction is completed. By poking a certain location before calling the PICDRAW routine, you can put it in single-step mode. You must use the normal PICDRAW call the first time, which will clear the screen and set the single-step mode, and use a second call to execute a step in the picture file. Repeated calls to this second location will eventually draw the entire picture. You can use this in a delay loop to slow the routine down, to vary the speed at different points, or whatever you like. The following are the pokes and calls necessary for each version of PICDRAW. Of course, you must poke in the starting address of your picture file as always, and the necessary X, Y location if needed for drawing an object over the picture.

For PICDRAWH use: POKE 36124,1 to set single step
CALL 36096 to clear the screen
CALL 36210 to execute one step in the picture
**Executing POKE 36124,0 will return you to normal, full speed drawing.

For PICDRAWL use: POKE 2076,1 set single step
CALL 2048 clear screen
CALL 2162 execute one step in the picture
**Executing POKE 2076,0 will return to full speed drawing again.

Memory Usage and Different Versions of PICDRAW

PICDRAWH

PICDRAWH takes 3K bytes and resides in locations 36096 to 39167 ($8D00 to $98FF). Figure 13.2 shows a memory map of where everything is located when using PICDRAWH. Your picture buffer (where the picture commands get loaded) may be placed anywhere where there is room, but the usual place is directly below the PICDRAWH routine. To find the highest location you may use for the buffer, find the length of your longest picture (from the picture editor or the binary transfer utility) and subtract it from 36096 (the starting location of PICDRAWH). Sometimes it is useful to actually use two picture buffers. You may want one for "room", or background, pictures, and another separate buffer for objects.

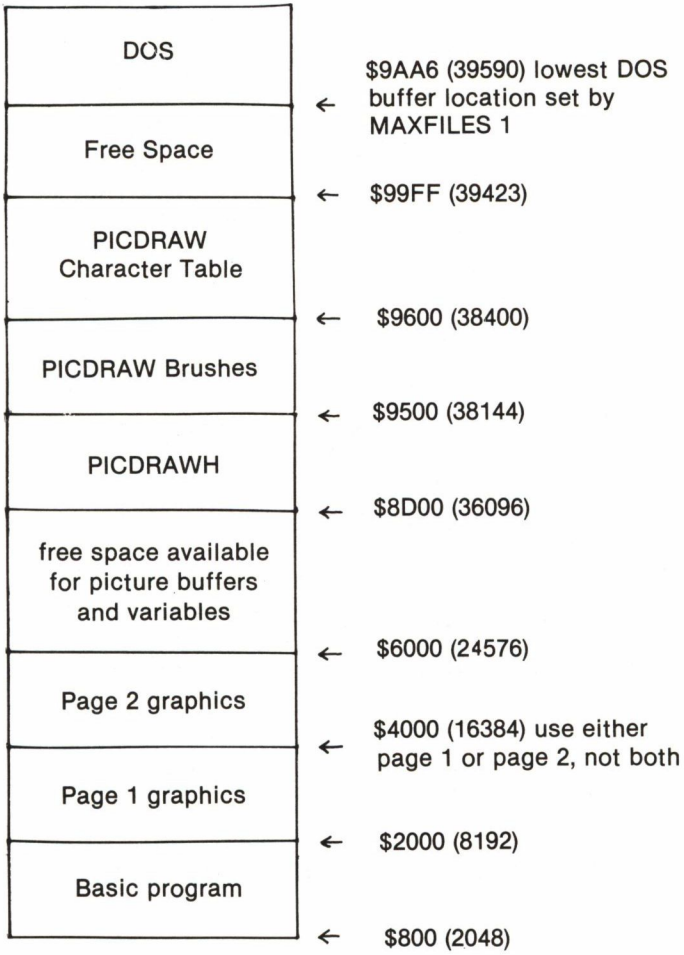
PICDRAWH requires that a MAXFILES 1 command be issued before it is loaded. This frees 1190 extra bytes, of which 768 are used by the text character table. To set MAXFILES 1 from a program, the following should be your first command:

```
1 PRINT CHR$ (4) "MAXFILES 1"
```

You should also use a HIMEM command to protect the PICDRAW routine and your picture buffer from your program. The command looks like:

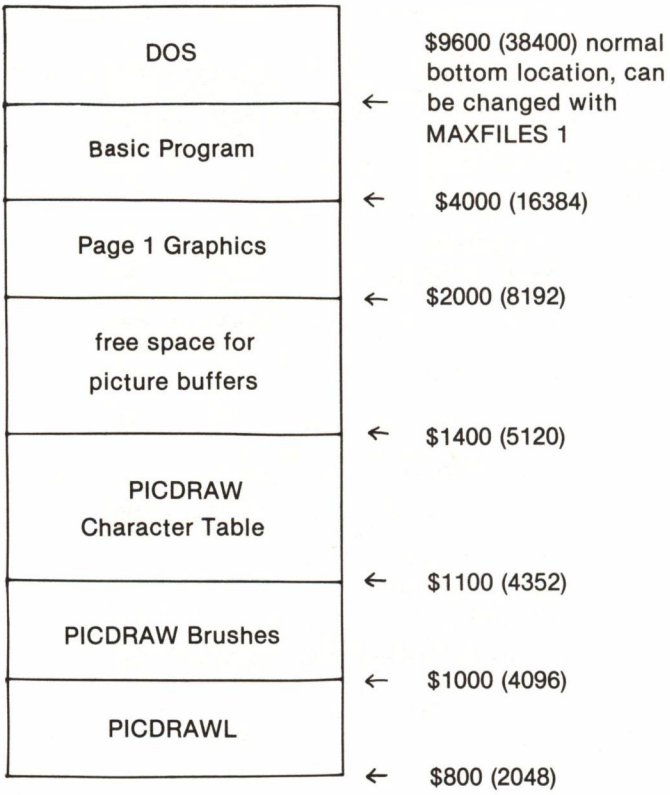
```
2 HIMEM: a
```

where "a" is the address of your picture buffer, above which everything should be protected.



PICDRAWH also uses zero-page locations $0-$C, $FC-$FF (0-12, 252-255)

Figure 13.2 - PICDRAWH Memory Map



PICDRAWL also uses zero-page locations $0-$C, $FC-$FF (0-12, 252-255).

Figure 13.3 - PICDRAWL Memory Map

PICDRAWL

PICDRAWL is located at the bottom of memory, where Applesoft programs usually start. Since there is a lot more memory above the hi-res pages than below, sometimes it's useful to rearrange memory so that the PICDRAW routine and picture buffers are below graphics page 1 and a long program starts after graphics page 1. Figure 13.3 shows the resulting memory usage.

PICDRAWL starts at location 2048 ($800) and goes up to location 5119 ($13FF). That means 3072 bytes are free for buffers starting at location 5120. If the text table is removed, as described in the next section, you can start your buffers at 4352.

The CALL and POKE locations for PICDRAWL are all 34048 less than those for PICDRAWH (it's been moved down 34,048 bytes for the lower location). The only exceptions are locations 0 and 1, which are still used for the starting address of the picture buffer.

If you use PICDRAWL with a Basic program, you must use a trick to get the Basic program to load above hi-res screen 1 in memory. To locate a program starting at 16384 ($4000), the first location above graphics page 1, you must write another program that will do a couple pokes, then run your program. This "loader" program is shown in listing 13.6. Substitute the name of your program for "name".

```
10 REM * O.K. TO LOAD COVER   PICTURE HERE *
20 POKE 103,1: POKE 104,64: POKE 16384,0
30 PRINT CHR$ (4);"RUN NAME"
```

Listing 13.6 - Loading a Basic Program above Graphics Page 1

Removing the Character Table

If you do not use the text option in any of your pictures, you can remove the text character table from PICDRAW and save 768 bytes. For PICDRAWH that means that you would not need a MAXFILES 1, or that you could use that space for something else. For PICDRAWL, it means that you could move your buffer down 768 bytes or again use that area for some other purpose.

To make a version of PICDRAW without the text table, use one of the following:

MAXFILES 1

BLOAD PICDRAWH

BSAVE PICDRAWH.NT,A$8D00,L$900

or:

BLOAD PICDRAWL

BSAVE PICDRAWL.NT,A$800,L$900

The ".NT" means "no text", and will help you remember that it is not a complete version.

If all your pictures use neither text nor brushes, you can eliminate another 256 bytes for the brush table. In the above, change "L$900" to "L$800" to take 256 bytes off the top. Also change the ".NT" to ".NB", or something appropriate.

Changing the Brush and/or Character Table

The brush and/or character table may be changed if you have a small font editor such as that available with **The Complete Graphics System** from Penguin Software or **The Applesoft Tookit** from Apple Computer. Several alternate text fonts are also available in **Additional Typesets** from Penguin Software.

The eight brushes are actually part of the text character set. They correspond to character numbers 0-31, which are normally non-printing control characters. Each brush is made of a group of four characters. The first brush, for example, is made of characters 0-3, with 0 the upper left corner, 1 the upper right corner, 2 the lower left corner, and 3 the lower right corner. The text characters correspond directly to the actual characters as they would appear in any font editor.

To load an alternate text set and keep the same brushes, use the following steps:

1) Create a PICDRAWH.NT or PICDRAWL.NT file as described in the previous section.

2) Type MAXFILES 1

3) If using PICDRAWH, type

BLOAD textset,A$9500

where "textset" is the name of the replacement character set. If using PICDRAWL, replace "$9500" with "$1000".

4) BLOAD PICDRAWH.NT

or BLOAD PICDRAWL.NT

5) BSAVE PICDRAWH, A$8D00,L$C00

or BSAVE PICDRAWL,A$800,L$C00

In step 5, you may want to use an alternate name so that you recognize it as one with a modified character set.

If you wish to replace both the text set and the brushes, simply reverse the sequence of steps 3 and 4.

Chapter 14 - The Picture Lister

Pressing "(**L**)ist picture" from the menu gets you to a picture listing utility. Normally, this will be of little use, but it does have a few interesting applications.

First, you can print out on your printer a list of commands that verbally describe a picture. These are the same commands that are listed individually at the bottom of the screen while you are editing a picture.

More importantly, the Lister lets you list your picture commands to a text file. Normally a picture's commands are stored in a binary file for speed. When the commands are taken apart and listed to a text file, a few things can happen. First, it becomes much easier to transfer pictures via modem to other Apple computers. Second, it becomes possible to transfer your pictures to other types of computers, and for them to interpret the pictures in a way best suited for display on that system. Third, it becomes possible to reload the picture on your own Apple, altering various parameters in the translation back to binary format.

Simple Options

Briefly going through the options "(**L**)oad" and "(**S**)ave" let you load and save sequential pictures from **The Graphics Magician** to and from disk. "(**P**)rint picture commands" lets you print the instructions that make up a picture to your printer. "(**V**)iew picture", "(**D**)isk catalog", and "(**Q**)uit" do just as they say.

"(**W**)rite text file" takes the current picture loaded in memory and puts its commands into a **Graphics Magician** picture transfer format. The commands are written out so that they can be interpreted by other Apples, and by other computers such as those from Atari, Commodore, and IBM.

Reading and Interpreting Transfer Files

"(**R**)ead text file" lets you read and interpret a picture transfer formatted file. This file can be created with an Apple, as in above, or it could be a graphics file created with another computer. In either case, you are first asked to enter a scaling factor. Apple files are written out with a scaling factor of 10; that is, all x,y coordinates are multiplied by 10 before being written to the transfer formatted file. To read an Apple picture back in, you would want to divide each coordinate by 10, therefore enter 10 for the scale factor.

Other computers do not necessarily have the same 280 dot by 192 dot resolution of an Apple. Files written out by other computers may use different scaling factors, or require that different scaling factors be used for interpreting their files so that the end result is a picture that is proportional to a full screen on an Apple.

You may also use the scaling factor to play tricks with the physical size of pictures created with an Apple, making them larger or smaller. One thing to remember is that the brush and text sizes do not change, so whereas the line and fill commands will actually condense or expand properly in size, brushes and text will be the same size, but move apart or scrunch together.

For those interested in playing around with the actual code for interpreting the pictures, instructions for reading and interpreting a picture file are in lines 500-600 of the program PICDOC, with the specific commands for interpreting the coordinates at line 525. You can do some pretty weird and interesting things by playing with that part of the code.

Color Table

The other interpretation that occurs when reading a file is that of color. Different types of computers differ greatly when it comes to color availability, so a method is provided to interpret colors as best possible across systems. When the interpreter reads a color from the color table, instead of automatically assigning that color number, it looks in a color table to find the equivalent that should be used.

The existing color table uses 256 color equivalences, with each number equivalent to itself (in other words, color 12 is interpreted as a 12). Colors 108-255 are not used on the Apple, but may be on other systems, hence the capability is included for interpreting them as best possible on the Apple.

Pressing "(**E**)dit color table" will give you a set of choices that allows you to load or save a color table of your choice, print the current color table equivalences to a printer, load the standard Apple equivalence color table (which is loaded in automatically when the program is run), change any one of the color equivalences, or return to the main options.

If you choose to change a color equivalence, you'll be asked which number you want to change, and what you want it to be. If you choose to change 5 to 17, whenever a picture transfer file is read in and interpreted, any color 5 will be interpreted as a 17. Besides being irreplaceable as a tool for transferring colored pictures between two different types of computers, this will also allow a quick way to change colors in any picture created on an Apple. It lets you bypass the need for going into the picture editor and single-stepping to find all the color commands that you want changed. The whole step is done automatically when you write a picture transfer file, change the color equivalences, then read and interpret the same file with the new colors. In fact it's so quick that it allows you to easily test several different color combinations for the same picture very painlessly.

Chapter 15 - Extras

The Binary Transfer Utility

Choosing "(**B**)inary transfer" from the menu lets you use the binary transfer utility. On the Apple, there are four types of storage files, designated by a letter in front of the name when you do a disk catalog, "A" files are Applesoft Basic files. "I" files are Integer Basic files (not used much anymore). "T" stands for text file, and "B" is a binary file.

BASIC files are easy to move around; you just LOAD them and SAVE them. The computer does all the work with location and length. All the files you create with **The Graphics Magician**, though, are binary files. They either contain binary data or machine language code. To move a binary file, you must know its starting address and length (in bytes). The binary transfer utility lets you easily find these. It will also let you automatically load a binary file from one disk and save it to another disk.

The options from the binary transfer utility are to "(**L**)oad", "(**S**)ave", "(**D**)isk catalog", or "(**Q**)uit". Each time you load a binary file, its starting address and length are automatically displayed, both in decimal and hexadecimal. Once you load a file, you don't necessarily have to save it. You would do so only if you want to put it onto another disk.

With a couple commands from Applesoft it is possible to change the starting location (and thus, loading location) of a binary file. Generally you should not do this with machine language program files, since these are usually dependent on location. You may easily make the change with binary data files, such as pictures, shapes, and paths, though. If you have a sequential picture called HOUSE, for example, that was 387 bytes long and you want to change its loading address to 24800, you would use the following from the Applesoft prompt (]):

```
BLOAD HOUSE.SPC,A24800

BSAVE HOUSE.SPC,A24800,L387
```

Demo

A demonstration of some of the features of **Graphics Magician** along with some sample picture and animation files is provided on the data side of your disk. To see it, type RUN DEMO from Applesoft.

Animated Alphabet

The 26 characters of the alphabet, all designed as animated shapes, are available on the data side of the diskette as well. You may use them in the animation editor, or modify them with the shape editor. Each is stored under its letter name.

Hi-res Text Generator

A small character generator has been included that will let you print text on the hi-res graphics screen from your programs. It is located at the very top of memory and requires a MAXFILES 1 command to be issued before the routine can be loaded and used. There are four versions on disk:

HPRINT ANIM for use with the animator (starts at $9700, or 38656)

HPRINT PH for use with PICDRAWH (starts at $9A00, or 39424)

HPRINT PL for use with PICDRAWL (starts at $9A00, or 39424)

HPRINT ALONE for use with none of the above (starts at $9500, or 38272)

The routine itself will simply plot an ASCII character in the range 32-127 at a specified x,y coordinate. Y may be anywhere in the range 0-184, X must be 0-39. To use the routine, first load it, then, given an x,y coordinate and an ASCII value A, put the ASCII value in 39424, put X in 39425, put Y in 39426, and CALL 39427. (Hexadecimal addresses $9A00-$9A03). A sample is shown in listing 15.1, and an ASCII table can be found in your Applesoft manual. The numerals 0-9 have ASCII values of 48-57, and the letters A-Z have values 65-90.

```
5 HOME
10 PRINT CHR$ (4);"MAXFILES1"
15 HIMEM: 38272
20 HGR
30 PRINT CHR$ (4);"BLOAD HPRINT ALONE"
40 GET A$
50 A = ASC (A$)
60 IF A = 13 THEN Y = Y + 1:X = 0: IF Y > 19 THEN Y = 0: GOTO 40
65 IF A = 8 THEN X = X - 1: IF X < 0 THEN X = 0
70 IF A = 27 THEN END
75 IF A < 32 OR A > 127 THEN 40
80 X = X + 1: IF X > 39 THEN X = 1:Y = Y + 1: IF Y > 19 THEN Y = 0
90 POKE 39424,A: POKE 39425,X: POKE 39426,(Y * 8)
100 CALL 39427
110 GOTO 40
```

Listing 15.1 - Using HPRINT

The versions of the text generator that work with PICDRAW use the same character set used by PICDRAW. The character set for the other two versions is located from 38656-39423 ($9700-$99FF, and can be replaced with any other standard small character set created in **The Complete Graphics System** or **The Applesoft Toolkit,** or from **Additional Typesets.** When using HPRINT ALONE, you should also use the command HIMEM:38272.

Capturing Shapes and Extra Editing

Pre-shifted shapes have their own special format and must be created with **The Graphics Magician** shape editor. There are ways, however, to convert standard Applesoft shapes or parts of a hi-res screen to pre-shifted shapes. The trick is to put those shapes on the screen in the editing area used by the shape editor, then jump into the shape editor and convince it that the shapes belong there.

There are three extra programs on your **Graphics Magician** disk that let you do this. SHAPE START lets you convert Applesoft shapes to pre-shifted shapes. SHAPE CAPTURE allows you to convert any part of a hi-res graphics screen into a shape. And with SHAPE SCREEN START you can edit pre-shifted shapes with any other graphics utility, then return to the shape editor.

To use SHAPE START, quit **The Graphics Magician** and type "RUN SHAPE START" (without the quotes). You will be asked for the name of your Applesoft shape table (give the entire name as it appears in the disk catalog), and it will be loaded from your data disk. You'll then be asked for the number of the shape that you want converted, and the border size that you want around your pre-shifted shape. The borders will be drawn, and you can position your shape within them by using the **IJKM** keys. "(**N**)ew" lets you choose a different shape in the same table. When the shape is positioned in the boxes, press "(**S**)hape editor" to jump into the editor. You'll have to tell it the border and shape size, as if you were starting new, but your shape will already be on the screen.

SHAPE CAPTURE works similarly. Quit **The Graphics Magician** and type "RUN SHAPE CAPTURE". You will be asked for a picture name. The picture must be stored in standard ".PIC" format. Type its name (without the ".PIC") and it will be loaded from your data disk. When the picture appears, use the **IJKM** keys to move the cursor around. When you reach the upper left corner of the area you want captured, press "1". Then move to the lower right corner and press "2". The bordered area will now be transferred seven times to an editing screen, and the shape editor will be run. As with SHAPE START, you'll have to give border, shape size, etc., as if you started new, but the shape will already be on the screen.

SHAPE SCREEN START lets you jump out of the shape editor, edit the "edit screen" with another graphics utility, then reenter the shape editor. It is a three-step process. First, when you have a shape on the shape editor "edit screen" (seven shapes, with border dots, etc.), go to the "(**O**)ptions" and press **Control-S** (an unlisted option; hold down the CONTROL key and press "S"). This saves the edit screen intact as a standard 34-sector ".PIC" file on your data disk, with the name you give it. Step two is to edit this picture file however you want with the graphics utility you desire. The only important part of this step is to erase the border dots that were left by the shape editor before you are done and proceed to step three. The last step is to "RUN SHAPE SCREEN START", give the name of the edited picture file, and you'll be put back into the editor, just as with the two previous programs.

Appendix A - Apple Colors

To fully take advantage of the color graphics of the Apple, it helps if you have an understanding of how colors are stored on the Apple graphics screen. Figure A.1 shows a magnified portion of a graphics screen. Each dot corresponds to a dot that may be on or off on the screen, and each is technically stored as a bit (on or off). The lines divide the basic storage units on the Apple, bytes. Each byte consists of seven bits displayed horizontally on the screen. An eighth bit is used as a color flag, or switch, which we will find is the key to most color anomalies on the Apple. The screen is 40 bytes, or 280 dots, wide. It is 192 bytes, and dots, tall.

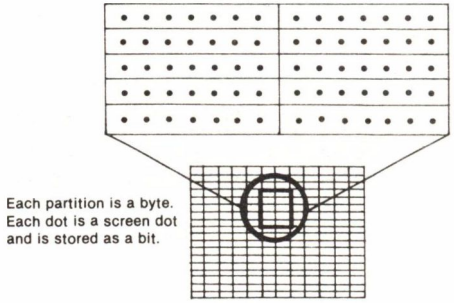


Figure A.1 - Section of Apple Graphics Screen

Each dot may only be on or off. If a dot is off, it is black. An "on" dot can be one of four colors, depending on two conditions. "On" dots in even columns will be either blue or violet. "On" dots in odd columns will be either orange or green. If the color flag for the byte is on, the dots will be orange and blue, and if the flag is off, the dots will be green and violet. Got all that? Okay, now if two dots are "on" horizontally next to each other, they both become white. This gives the "eight" Apple colors, even though there are two versions of white, and two versions of black. See figure A.2.

	Group 1	Group 2	DOTS BY COLUMN
	Color Flag Off	Color Flag On	even odd even odd even odd even
Dots Off	Black #0	Black #4	○ ○ ○ ○ ○ ○ ○
Odd Dots On	Green #1	Orange #5	○ ● ○ ● ○ ● ○
Even Dots On	Violet #2	Blue #6	● ○ ● ○ ● ○ ●
All Dots On	White #3	White #7	● ● ● ● ● ● ●

on ● off ○

Figure A.2 - Color Construction

The color layout has a few interesting results. First, black and white mode is basically dot on or dot off, without regard to color flag or position. "White" means a dot is on, and it can actually be displayed as white, blue, violet, green, or orange. "Black" means the dot is off, and will always be displayed as black. The actual resolution in this mode is 280 by 192. (Note: the color flag also slightly affects the position of a dot, shifting it left or right a half position. By using this shift, it is possible in some instances to make the horizontal resolution appear to be 560 since there are 560 positions in which dots can appear. It is not a true resolution, however, since no matter what you do there are only seven dots per byte.)

The second result of the color layout is that in color mode, the resolution is really only 140 dots wide. An orange line across the screen has only the odd dots set; the even dots must be off. It only uses 140 dots and restricts the use of the others. This is part of the reason why most uses of color have a black background. Any color may be used against a black background, although in limited positions. Against any other background, even white, you are restricted to a much lower apparent resolution.

The third result is that some colors are difficult to use next to others because of the color flag. Orange and green dots, for example, cannot appear in the same byte, so it's usually impossible to use them next to each other horizontally. Looking at figure A.2, the colors in group 1 generally cannot be used horizontally next to the colors in group 2.

Blended Colors and the Color Palette

There are ways to combine colors so that you perceive different shades and textures, and even get fooled into thinking there are a few new colors. Part of it is just creating different color patterns horizontally. The other part is that colors that cannot be placed horizontally next to each other can be placed in alternate rows vertically. Green and orange, for example, when placed in alternate rows give a yellow color.

The color palette available in the picture editor actually has three groups of colors. Group A, the first and largest set, consists of blended colors such that alternate rows use group 1 and group 2 standard Apple colors (as in figure A.2). Group A is the first 5 columns of the palette and the top two colors from the sixth column, or colors 0 to 51. (The first color in each group is white).

Group B consists of blends of colors made only with the Apple colors 4-7, and is colors 52 to 76 on the palette. Group C is made of Apple colors 0-3, and is colors 77 to 107 on the palette. Lines from the matching set of line colors can be used over group B or C, but lines should not be drawn over group A colors. Instead, the lines should be drawn first, then colors added.

"Color bleed" is when color flags change around the border of two colors and cause the first color to change at the border. In the picture editor, color bleed may be minimized by avoiding putting two colors from different groups next to each other horizontally. Instead, work in horizontal zones that keep the conflicting colors apart horizontally, but allow them to share vertical borders (see figure A.3). Note also that several colors appear in two or three of the groups. They are constructed differently, but appear the same. There are three different whites, for example. You can use these different colors across the zones to make it appear that one color spans them (see figure A.4 for an example).

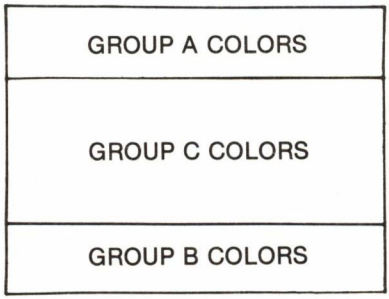


Figure A.3 - Color Zone Example

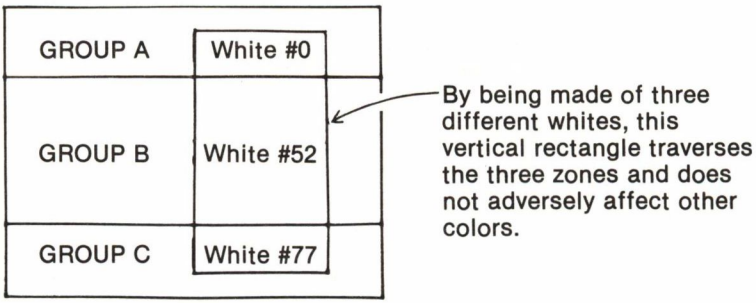


Figure A.4 - Traversing Zones

Appendix B - Filename Suffixes

Since there are so many types of graphics files that can be created with our various graphics products, we've added an identifying suffix to each one so that when a disk is cataloged the file type may be easily known. Here's a list of the types of data files that can be created with **The Graphics Magician**:

.ANM binary animation and machine language file created with the animation editor

.ATX animation text file saved with the animation editor; not necessary for running the animation, but needed for re-editing the animation file or documenting it with the animation documenter

.FNT text character font created with **The Complete Graphics System** text editor, or available with **Additional Typesets;** the small fonts may be used with the PICDRAW and HPRINT routines

.PIC standard format 33 or 34 sector screen picture image, loads on graphics page 1, and saved with the "(**I**)mage" command in the picture editor or with any of our other graphics software.

.PTH binary path table, created with the path editor.

.PTX picture text file; a transfer formatted sequential picture saved with the Picture Lister.

.SPC sequential picture file created with **The Graphics Magician** picture editor.

.SSH binary pre-shifted shape file created with **The Graphics Magician** shape editor.

.STS small typeset, same as .FNT, but distinguishes between small and large typesets.

Appendix C - Notes for users of the original version of Graphics Magician

Most of the machine language routines have been changed to add new features or further optimize previous ones. Some of the data file structures have also been modified to allow extra features or save space. You should carefully read and note all the new entry points and tables for the machine language routines, and check the new features available; they may save you some time and code.

Animation System

The shape format has been shortened slightly (12 bytes per shape). Old shapes can be converted to the new format using the SHAPE CONVERTER program provided.

The path format has also been changed to allow for moves of +-7 instead of +-3. If you enter any values directly into your path from a program, you should find where that is done and compute new values. The old path values are changed. You may convert old paths to the new format with the PATH CONVERTER program provided.

The animation editor puts out a differently formatted file with some new, added information besides having the new format shapes and paths, so any animation files created with the original version will have to be recreated with the new.

The CALL locations have changed for the animation machine language routines. Tables, while holding mostly the same information, have been moved around and split to save computation time in the animator and in your program. Check chapters 8 and 9 carefully, if you are modifying a program that already uses the old animator to use the new. Also note the new animation types available.

Picture System

The sequential picture files are in the same format as the original files, and may still be used. The text commands are new commands and do not affect readability of the older files. The fill routine has had a slight correction made in the averaging routine and some older pictures actually rely on the original version to fill a section properly. You should check older pictures to see if they fill totally with the change made. If they don't, you may have to go into the editor and add an extra fill command or two.

The POKE and CALL locations for the PICDRAW routines have changed. The routines in this version are 256 bytes shorter and faster than PICDRAWF in the original version. They are also 256 bytes longer than the original, much slower PICDRAW. On top of that, however, the 768 optional bytes of character tables for the text routine have been added.

Appendix D - Reference Card for Shape Editor, Path Editor, and Picture Editor

Shape Editor

I,J,K,M cursor movement up, left, right, and down

Z	plot one point
X	erase one point
Q	lock plotting
W	lock erasing
C	set color
B	reset borders and boundaries
F	flip shape
R	rotate shape
S	shift shape
Control-S	shift shape outside boundaries
E	exchange colors
A	animate
1-7	turn on and off frames 1-7
O	options to disk:

E	go back to edit mode
S	save shape
Control-S	save shape editor screen
L	load shape
D	disk catalog
C	clear shape
Q	quit

Path Editor

U,I,O,J,K,N,M,","," cursor "move and plot" keys

RETURN, ←, →,/
or cursor move keys, one dot each
//e arrow keys

Z	plot move
D or DELETE	delete the last step
X,Y	change standard x,y moves
ESC	show full screen
S	save path
C	clear path
Q	quit

Picture Editor

T	enter text mode at cursor position
Control-T	enter text mode at last text position
Z	zero in on area
Control-Z	control how tight "Zero mode" is
SPACE BAR	go to selection page, or back to picture page
ESC	full screen switch
R	redraw picture, or leave edit mode
D or DELETE	delete the last step
Control-D	delete multiple steps
E	enter edit mode at beginning of picture
←	enter edit mode from end of picture
S	save sequential picture
I	save screen image
Q	quit

Text Mode Commands

Control-L	normal/lower keyboard toggle
Control-R	reverse type
Control-N	normal colored type
ESC	full screen switch
Control-D, DELETE, or ←	delete last letter
RETURN	leave text mode

Edit Mode Commands

Same as normal commands, with these added:

→	advance one step
←	go back one step
> or TAB	tab forward 10 steps

Unlisted Commands

J	reverse joystick orientation
L	line mode, without going to selection page
F	fill mode, without going to selection page
1-8	brush selection, without going to selection page

Put professional graphics into your own programs!

The Graphics Magician contains machine language animation routines that use the same techniques as most popular computer arcade games. Three animation editors let you design figures, their paths, assemble animations with up to 32 independent objects, and then control the animation from your own programs.

A high-resolution picture/object builder is included that lets you store hundreds of multi-colored pictures on a single disk and recall them quickly from your own programs. This capacity is useful in designing adventure games, educational software, and other programs requiring a multitude of easily accessible graphic images.

Design of graphics is done through menu-driven editors. To use in your own programs, just add our machine language routines to your programs. The entire package is designed to be easy enough for the beginning programmer, yet with the power and flexibility to satisfy the most advanced.

Softalk magazine's 1983 poll of the most popular software as determined by end-users showed The Graphics Magician to be:

*The most popular non-game program of 1982!
*The 19th most popular program of all time!

Controls:

Animation editing is done through the keyboard. Picture editing can be done with paddles, joystick, trackball, or Apple Graphics Tablet. Ask your dealer for information about this program's compatibility with input devices.

Other Penguin Software programs:

Short Cuts
The Complete Graphics System
Additional Type Sets
Map Pack
Transitions
Paper Graphics

The Coveted Mirror
Minit Man
The Spy Strikes Back
Pensate
The Quest
Thunderbombs
Crime Wave
Spy's Demise
Transylvania
Pie Man

The Apple version of The Graphics Magician is written by Mark Pelczarski and designed by Mark Pelczarski, Chris Jochumson and David Lubar.